

# Cours d'algorithmique

---

ESGI 2<sup>ème</sup> année

2021-2022

baudoin2020@gmail.com

## L'algorithmique à l'ESGI

- 1<sup>ère</sup> année : **brut force**
  - Itératif : imbriquer plusieurs boucles avec appel de fonction pour traitement
    - ✖ traitements : *filtrage, comptage*, calcul de min et de max
  - Récursivité
- 2<sup>ème</sup> année : **performance**
  - itératif
  - Récursivité
- 3<sup>ème</sup> année : tris et structures : implémentation

## Comment mesurer la performance d'un algorithme ?

- calcul du temps d'exécution → performance **empirique**
- calcul de la complexité → performance **théorique**

## Comment améliorer la performance d'un algorithme ?

- Utilisation de *Sets* et de *Hashmap*
- Récursivité : Backtracking
- Méthodes générales
  - Diviser pour régner
  - Programmation dynamique
- Méthodes ad-hoc
  - balayage, matrice des correspondances, ...

## Progression

### Séance 1

- introduction à python
- mesure du temps d'exécution (code timing) : un premier exemple
- *Filtrage* : Rechercher une valeur dans un tableau
  - recherche simple
  - recherche dichotomique
  - recherche dans un hashSet (ensemble)
- *Comptage* : tableaux associatifs : les tables de hachage (hashmaps)

### Séance 2 : TP

- exercices basiques
- exercice de synthèse : indexation de documents

### Séance 3

- CC1
- Calcul de complexité théorique
- implémentation hashset

### Séance 4 : Récursivité

- rappels
- backtracking

### Séance 5 : Récursivité : applications

- carré magique
- sudoku
- mots croisés

## Installation de Python

### Python3 avec au choix

- l'IDE natif IDLE
- votre Ide préféré (pycharm, atom, ...)
- des notebook jupyter, à installer
  - via winpython
  - ou via conda

## 1. Introduction à Python

### Variables, type, print et input, if

```
x=9
print(type(x))
```

```
<class 'int'>
```

```
x='toto'
print(type(x))
```

```
<class 'str'>
```

```
print(x)
```

```
toto
```

```
x=input('Entrez un mot : ')
print(type(x), x)
```

```
Entrez un mot : 2
```

```
<class 'str'> 2
```

```
x=int(input('Entrez un nombre : '))
print(type(x), x)
```

```
Entrez un nombre : 2
```

```
<class 'int'> 2
```

```
x=int(input('Entrez un nombre : '))
if x == 0 : print("-> nombre nul")
else : print("-> nombre non nul")
```

```
Entrez un nombre : 3
```

```
nombre non nul
```

```
for x in range(3, 7):
    print(x)
```

```
3
```

```
4
```

```
5
```

```
6
```

### Boucles

La fonction *range* et l'opérateur *in*

```
print(range(5))
```

```
range(0, 5)
```

```
print( 3 in range(5))
```

```
True
```

```
print( 3 in range(2))
```

```
False
```

la boucle *for*

```
for x in range(3):
    print(x)
```

```
0
```

```
1
```

```
2
```

```

: for x in range(3, 7):
    print(x)

```

```

3
4
5
6

```

### Exercice 1

- Afficher les nombres de 1 à 10
- Afficher la somme des nombres pairs entre 2 et un entier saisi par l'utilisateur

## Tableaux (listes)

```

: liste = []
: liste.append(3)
: liste.append(5)
: print(liste)

```

```

[3, 5]

```

```

: liste = [0]*10
: print(liste)

```

```

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

```

```

: liste = [3, 5, 2]
: print(liste[1])

```

```

5

```

```

liste.sort()
print(liste)

```

```

[2, 3, 5]

```

```

print(len(liste))

```

```

2

```

## Fonctions

```

def afficheListe(tab):
    for i in range(len(tab)):
        print(tab[i])

tab=[4, 6, 5]
afficheListe(tab)

```

```

4
6
5

```

### Exercice 2

Ecrire (et tester) les fonctions suivantes :

- `rechDoublons` qui indique (en renvoyant un booléen) si des doublons sont présents dans le tableau `tab`

- rechDoublonsVoisins qui indique si des doublons sont présents côte à côte dans le tableau tab

### Timing de code

```
import time

start = time.time()
# code à timer
s=0
for i in range(100000): s+=s*i
#
end=time.time()
print(end-start)
```

0.03296184539794922

```
import time

for taille in [1000,10000,100000]:
    start = time.time()
    # code à timer
    s=0
    for i in range(taille): s+=s*i-i*i
    #
    end=time.time()

    print('taille = ', taille, ' en ', end-start, 'sec')
```

taille = 1000 en 0.000995635986328125 sec  
 taille = 10000 en 0.09894180297851562 sec  
 taille = 100000 en 7.85050630569458 sec

```
import random
min=0
max=21
for i in range(10):
    print(random.randint(min, max)) #donne une valeur entre 0 et 20 (inclus)
```

19  
 5  
 12  
 13  
 5  
 18  
 12  
 2  
 12  
 5

### Exercice 3 Timing de code : méthodologie

#### Le pire des cas

Quand on souhaite mesurer le temps d'exécution d'une fonction, il convient de s'assurer que la fonction ne se termine pas prématurément.

Pour avoir un résultat significatif, il faut donc choisir des paramètres qui correspondent au *pire des cas*, c'est-à-dire au traitement le plus long.

Mesurer les temps d'exécution des fonctions **rechDoublonsVoisins** et **rechDoublons** sur des tableaux de taille 100, 1000, 10000 et 30000 dans le pire des cas

## 2. Rechercher une valeur dans un tableau

Cette opération fait partie des opérations les plus fréquentes dans une application : rechercher si un tableau contient une valeur donnée.

- la recherche simple : si on ne dispose pas d'information à propos des données du tableau
- la recherche dichotomique : si le tableau est trié

### Exercice 4 : recherche simple

Ecrire une fonction `rechSimple` qui prend en paramètre un tableau `tab` et une valeur `val` et qui renvoie un entier qui vaut :

- l'index de la première occurrence de `val` dans le tableau
- -1 si `val` ne se trouve pas dans le tableau

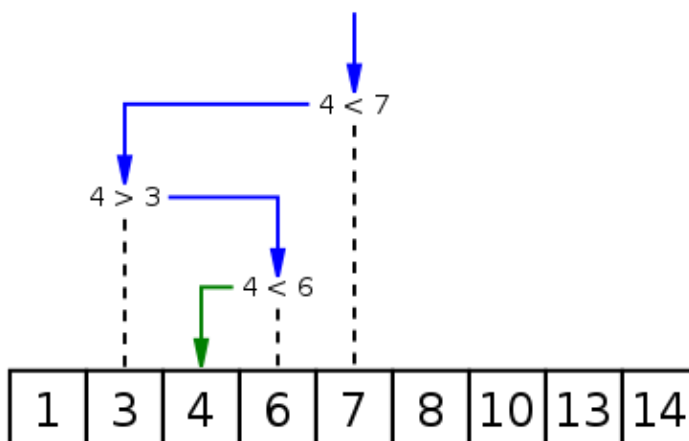
### Exercice 5 : recherche dichotomique (binary search)

Ecrire une fonction `rechDich` qui prend en paramètre un tableau `tab` **supposé trié** et une valeur `val` et qui renvoie un entier qui vaut :

- l'index de la première occurrence de `val` dans le tableau
- -1 si `val` ne se trouve pas dans le tableau

. Vous procéderez selon le principe suivant :

1. On tient en permanence deux indices gauche et droite tels que  $T[\text{gauche}] \leq \text{val} \leq T[\text{droite}]$ .
2. À chaque itération, on calcule l'index au milieu de l'intervalle [gauche, droite] et on compare `val` avec la valeur du tableau à cet index.
3. En fonction du résultat, soit on affiche l'indice du milieu, soit on met à jour gauche ou droite et on continue à chercher `n` dans le sous-tableau correspondant.



### Exercice 6 : Comparaison des fonctions de recherche

Mesurer les temps d'exécution des 2 fonctions de recherche pour 1 opération de recherche sur des tableaux de taille 100, 1k, 10k et 100k dans le pire des cas

### 3. Utilisation de HashSets

Les *sets* (ensembles) sont des structures de données destinées à accélérer les opérations de recherches.

#### Set en python

Initialisation

```
: mySet={10, 20}
  print(mySet)
  #ou
  mySet = set()
  print(mySet)

{10, 20}
set()
```

Ajouter un élément.

```
: mySet.add(2)
  print(mySet)

{2}
```

Pour ajouter les éléments d'une liste.

```
: mySet.update([5,3,4])
  print(mySet)

{2, 3, 4, 5}
```

Les set ne contiennent pas de doublons

```
mySet.add(2)
print(mySet)

{2, 3, 4, 5}
```



On peut retirer un élément d'un set avec `remove` (qui renvoie une erreur s'il ne s'y trouve pas) ou `discard` qui marche tout le temps

```
mySet.remove(1)
```

```
-----
KeyError                                Traceback (most recent call last)
<ipython-input-14-f21715c34701> in <module>()
----> 1 mySet.remove(1)

KeyError: 1
```

```
mySet.discard(2)
print(mySet)
```

```
{3, 4, 5}
```

### Recherche dans un set

Test d'appartenance d'un élément dans un set.

```
3 in mySet
```

```
True
```

### Construction d'un set à partir d'un tableau

```
s=set()
s.update([5,3,4,3])
print(s)
```

```
{3, 4, 5}
```

### Exercice 7 : Comparaison des fonctions de recherche

Construisez un set avec des valeurs aléatoires (100, 1k, 10k et 100k valeurs ) et rechercher une valeur dans le pire des cas. Comparez le temps d'exécution avec les 2 autres méthodes de recherche de l'exercice 6.

### Exercice 8 : Exercice avec les sets

Ecrire une fonction `rechDoublonsRapide` qui prend en paramètre un tableau d'entiers `tab` et qui renvoie un booléen qui indique si `tab` contient des doublons.

On utilisera un set auxiliaire, de manière à accélérer la recherche.

Indice : le set doit contenir les valeurs « déjà vues »

#### 4. Utilisation de Hashmaps

Les hashmaps (tables de hachage) sont un type de tableau associatif, une structure de données qui repose sur une correspondance clé-valeur.

| key      | values |
|----------|--------|
| 'toto'   | 3      |
| 'tata'   | 89     |
| 'titi'   | 45     |
| 'toutou' | 12     |

En python, les hashmap sont des dict (pour dictionnaires). Dans le cas du dictionnaire, les clefs sont les mots (définis) et les valeurs sont les définitions de ces mots.

- Les clés sont uniques et leur type est quelconque.
- l'ensemble des clefs forment un set

##### Initialisation

```
hm={ }
#ou
hm={'a':2, 'b':4}
print(hm)
hm['F']=5 # ajout d'une ligne
print(hm)
```

```
{'a': 2, 'b': 4}
{'a': 2, 'b': 4, 'F': 5}
{'b': 3, 'a': 3}
```

##### Recherche dans une hashmap

```
print('a' in hm)
print('r' in hm)
```

```
True
False
```

#### Exercice 9 : Exercice avec les hashmaps

Ecrire une fonction `rechTriplonsRapide` qui prend en paramètre un tableau d'entiers `tab` et qui renvoie un booléen qui indique si `tab` contient des triplons. Attention, vous pouvez utiliser une hashmap auxiliaire mais il est interdit de la parcourir

Ecrire également une autre fonction (brute force) et comparer les temps d'exécution sur des tableaux de tailles différentes.

## 5. Exercices de TP

### Exercice 1

On cherche à écrire une fonction `premCarRec(tableau)` qui renvoie le **premier élément récurrent** d'un tableau (dans une lecture des indices croissants). Il s'agit du problème du *First Recurring Character*

Version 1 : renvoyer l'élément présent plusieurs fois dans le tableau, dont on découvre la première occurrence en premier Par exemple,

- `tableau = [3, 6, 2, 7, 2, 6, 1]`, la fonction doit renvoyer 6

Version 2 : renvoyer l'élément présent plusieurs fois dans le tableau, dont on découvre la deuxième occurrence en premier Par exemple,

- `tableau = [3, 6, 2, 7, 2, 6, 1]`, la fonction doit renvoyer 2

Pour les 2 versions :

- `tableau = list('hippopotame')`, la fonction doit renvoyer 'p'
- `tableau = [3, 6, 2, 7, 1]`, la fonction doit renvoyer `None` car il n'y a pas de doublon

Version 1

1. Ecrire une solution de type **brut-force**
2. Ecrire une solution plus rapide **en utilisant une hashmap** et évaluer le gain de performance sur un gros tableau de taille >50k

Version 2

1. Ecrire une solution de type **brut-force**. Est ce possible en 2 boucles seulement ?
2. Ecrire une solution plus rapide **en utilisant un set** et évaluer le gain de performance sur un gros tableau de taille >50k

### Exercice 2

On cherche à écrire une fonction `premCarNonRec(tableau)` qui renvoie le **premier élément non-répété** du tableau (dans une lecture des indices croissants), c'est à dire celui qui ne réapparaît pas et qui est lu en premier

Par exemple,

- `tableau = [3, 6, 2, 5, 7, 2, 6, 3]`, la fonction doit renvoyer 5
- `tableau = list('hippopothame')`, la fonction doit renvoyer 'i'
- `tableau = [3, 6, 3, 6, 6]`, la fonction doit renvoyer `None` car il n'y a pas de valeur unique

1. Ecrire une solution de type **brut-force**
2. Ecrire une solution plus rapide, **avec une hashmap**

3. Ecrire une solution rapide avec **des sets**

### Exercice 3

On cherche à écrire une fonction **mostFreq** qui permet de déterminer l'élément le plus fréquent dans un tableau. En cas d'égalité, renvoyer l'élément qui apparaît en premier dans le tableau (le plus à gauche)

1. Ecrire une solution de type brute force
2. Ecrire une solution rapide

### Exercice 4

On cherche à écrire une fonction **doubDoub** qui permet de déterminer si le tableau en paramètre contient au moins 2 doublons, c'est-à-dire 2 valeurs qui se répètent chacune au moins 2 fois.

Par exemple, sur le tableau `tab=[2, 4, 2, 6, 4, 7, 4]` la fonction doit renvoyer `true`

et sur le tableau `tab = [2, 4, 12, 6, 4, 7, 4]` la fonction doit renvoyer `false`

1. Ecrire une solution de type brute force
2. Ecrire une solution rapide

## 6. Complexité théorique

### Définition :

L'analyse de la complexité consiste à étudier la quantité de ressources (par exemple de temps ou d'espace) nécessaire à l'exécution de cet algorithme.

#### La complexité s'exprime en fonction de la taille de l'entrée de l'algorithme

Deux approches :

- calcul de la complexité dans le pire des cas
- calcul de la complexité en moyenne

**Exemple :** recherche d'une valeur dans un tableau (une liste)

#### 1. Recherche simple dans un tableau de taille n

```
def rechSimple(liste, val):
    for i in range(len(liste)):
        if liste[i]==val :
            return i
    return -1
```

Ici, on cherche à exprimer le nombre d'itérations de l'algorithme

- en fonction de la taille de liste (notée  $n$ )
- dans le pire des cas, c'est-à-dire quand la valeur ne se trouve pas dans le tableau

➔ Il faut  $n$  itérations. Donc la complexité vaut  **$O(n)$**

## 2. Recherche dichotomique

```
def rechDich(liste, val):
    g=0
    d=len(liste)-1
    while(g<=d):
        m=(g+d)//2
        if(liste[m]==val):return m
        if(liste[m]<val):g=m+1
        if(liste[m]>val):d=m-1
        #print(g, d, m)
    return -1
```

Soit  $n$  la taille du tableau liste. On suppose que  $n$  est grand et qu'il peut s'écrire  $n = 2^k$

A chaque itération de l'algorithme, on écarte la moitié des possibilités, c'est-à-dire qu'on le divise par 2. Il faudra donc  $k$  itérations pour s'assurer que la valeur ne se trouve pas dans le tableau (le pire des cas).

Or  $n = 2^k \leftrightarrow k = \log(n)$

➔ Donc le nombre d'itérations dans le pire des cas vaut  $\log(n)$ , la complexité se note  **$O(\log(n))$**

### Les ordres de grandeur : la notation de Landau $O(n)$

Soit  $a(n)$  le nombre d'instructions exécutées dans le pire des cas, en fonction de  $n$  (généralement la taille du paramètre d'entrée de l'algorithme).

Alors, on dit que  **$a(n) \in O(f(n))$**  si

$$\exists C > 0, \exists n_0, a(n) < C * f(n)$$

ou

$$\exists C > 0, \lim_{n \rightarrow \infty} \frac{a(n)}{f(n)} < C$$

Par exemple

$$2n^2 \in O(n^2)$$

$$\frac{1}{100000}n^2 - 1000n - 2000 \in O(n^2)$$

$$2^n + 2n^2 \in O(2^n)$$

$$200000 \in O(1)$$

Quelques règles utiles :

- Les coefficients peuvent être omis :  $4n^2$  devient  $n^2$
- $n^a$  domine  $n^b$  si  $a > b$  : par exemple,  $n^4$  domine  $n^3$
- Une exponentielle domine un polynôme :  $3^n$  domine  $n^2$  (cela domine également  $2^n$ ) •
- De même, un polynôme domine un logarithme :  $n$  domine  $\log(n)$ . Cela signifie également, par exemple, que  $n^2$  domine  $n * \log(n)$

### Exercice 1

Pour chacun des codes suivants, donner le nombre d'itérations de l'algorithme, ainsi que la complexité simplifiée. Par exemple :  $3 * n = O(n)$

|  |   |
|--|---|
| <p>Code 1</p> <pre>for (int i = 0; i &lt; n; i++) {     // instructions } for (int i = 0; i &lt; n; i++) {     // instructions }</pre>   | <p>Code 2</p> <pre>for (int i = 0; i &lt; n; i++) {     // instructions     for (int j = 0; j &lt; n; j++) {         // instructions     } } for (int j = 0; j &lt; n; j++) {     // instructions }</pre>         |
| <p>Code 3</p> <pre>for (int i = 0; i &lt; n; i++) {     // instructions     for (int j = 0; j &lt; i; j++) {         // instructions     } }</pre>                                   | <p>Code 4</p> <pre>for (int i = 0; i &lt; n; i++) {     // instructions     for (int j = 0; j &lt; n * i; j++) {         // instructions     } }</pre>  |
| <p>Code 5</p> <pre>def f(tab):     n = len(tab)     for i in range(n):         for j in range(5 * n):             print(i + j)     for i in range(n):         print(i * i * i)</pre> | <p>Code 6</p> <pre>def f(tab):     n = len(tab)     for i in range(n):         for j in range(n):             g(i + j) def g(k):     for i in range(k):         for j in range(k * k):             print(i)</pre> |

Comparaison des complexités

| <b>log(n)</b> | <b><math>\sqrt{n}</math></b> | <b>n</b>  | <b>n log(n)</b> | <b>n<sup>2</sup></b> |
|---------------|------------------------------|-----------|-----------------|----------------------|
| 3             | 3                            | 10        | 33              | 100                  |
| 7             | 10                           | 100       | 664             | 10 000               |
| 10            | 32                           | 1000      | 9966            | 1 000 000            |
| 13            | 100                          | 10 000    | 132 877         | 100 000 000          |
| 17            | 316                          | 100 000   | 1 660 964       | 10 000 000 000       |
| 20            | 1000                         | 1 000 000 | 19 931 569      | 1 000 000 000 000    |

| Opérations<br>par<br>secondes | Taille du problème: 1 million |                 |                      | Taille du problème 1 milliard |                 |                      |
|-------------------------------|-------------------------------|-----------------|----------------------|-------------------------------|-----------------|----------------------|
|                               | <b>n</b>                      | <b>n log(n)</b> | <b>n<sup>2</sup></b> | <b>n</b>                      | <b>n log(n)</b> | <b>n<sup>2</sup></b> |
| <b>10<sup>6</sup></b>         | Secondes                      | Secondes        | Semaines             | Heures                        | Heures          | Jamais               |
| <b>10<sup>9</sup></b>         | Immédiat                      | Immédiat        | Heures               | Secondes                      | Secondes        | Décennies            |
| <b>10<sup>12</sup></b>        | Immédiat                      | Immédiat        | Secondes             | Immédiat                      | Immédiat        | Semaines             |

## 7. Implémenter ses propres sets et hashmaps

### Sets d'entiers

On souhaite rechercher des valeurs dans un tableau d'entiers (myTab), de taille quelconque, mais on veut éviter de parcourir tout le tableau à chaque recherche ;

- a. Version 0 : On suppose d'abord que tous les entiers sont dans un intervalle [0, 100]. On cherche à transformer le tableau myTab en un tableau mySet de taille 100 :

**mySet[i]=1 si i est une valeur du tableau myTab, et 0 sinon**

Plus simplement, si i est présent dans myTab alors mySet[i]=1

Par exemple (pour des nombres compris entre 0 à 10)

myTab : de taille quelconque mais avec 10 valeurs possibles (c'est-à-dire différentes) seulement.

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 4 | 7 | 9 | 4 | 7 | 9 | 3 | 3 | 5 | 5 | 5 | 5 | 5 | 5 | 2 | 7 | 4 | 7 | 7 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

mySet : de taille 10

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 |

- ➔ Ecrire une fonction `int[] tab2set0(int[] tab, int taille)` qui transforme myTab en mySet. (taille est la taille de mySet)

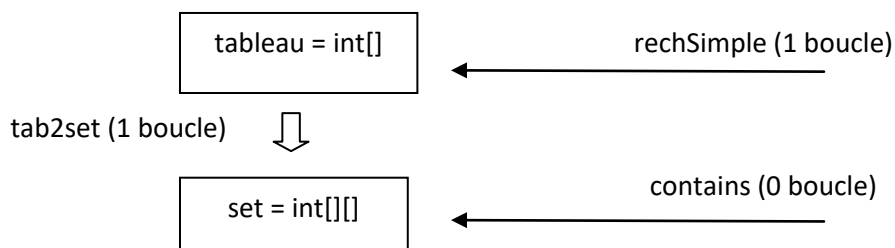
En python, votre fonction devrait commencer comme :

```
def tab2set0(mytab, taille):
    myset =[0]*taille #créé le tableau myset, initialisé avec des 0
    ...
```

- ➔ Ecrire une fonction boolean `contains0(int[] mySet, int val)` qui renvoie True si la valeur val se trouve dans myTab

```
def contains0 (mySet, val):
    ...
    ...
```

*Principe :*





- b. Version 1 : On suppose désormais que les valeurs de myTab sont dans un intervalle [0, 1000]. On pourrait créer un tableau MySet de taille 1000 mais on va continuer à utiliser un MySet de taille = 100. A la différence de l'exercice précédent, mySet ne sera plus un tableau d'entiers, mais un tableau de tableau d'entiers.

**si i est une valeur du tableau myTab,  
le (petit) tableau mySet[i%100] contient i**

Par exemple (pour des nombres compris entre 0 à 100 et un myTab de taille 10)

myTab :

|   |   |    |    |   |   |    |    |    |   |    |   |   |   |   |   |    |   |   |   |
|---|---|----|----|---|---|----|----|----|---|----|---|---|---|---|---|----|---|---|---|
| 4 | 7 | 29 | 14 | 7 | 9 | 33 | 93 | 85 | 5 | 25 | 5 | 5 | 5 | 2 | 7 | 14 | 7 | 7 | 7 |
|---|---|----|----|---|---|----|----|----|---|----|---|---|---|---|---|----|---|---|---|

mySet :

| 0  | 1  | 2   | 3        | 4           | 5                    | 6  | 7               | 8  | 9       |
|----|----|-----|----------|-------------|----------------------|----|-----------------|----|---------|
| [] | [] | [2] | [33, 93] | [4, 14, 14] | [85, 5, 25, 5, 5, 5] | [] | [7, 7, 7, 7, 7] | [] | [29, 9] |

→

- Ecrire une fonction `int[][] tab2set(int[] tab, int taille)` qui transforme le tableau myTab en double tableau mySet

En python, votre fonction devrait commencer comme :

```
def tab2set(mytab, taille):
    myset = [[] for i in range(taille)] #créé le tableau myset,
    initialisé avec des tableaux vides
    ...
```

Vous aurez également besoin d'ajouter un élément dans un tableau. en python, cela se fait avec `tableau.append(val)`

- Ecrire une fonction boolean `contains(int[][] mySet, int val)` qui renvoie True si la valeur val se trouve dans myTab

```
def contains (mySet, val):
    ...
    ...
```

## Set de String

On cherche maintenant à adapter nos solutions pour le stockage et la recherche de String. Pour cela, on va calculer un entier à partir d'une String, pour ensuite stocker la String comme cet entier.

- Ecrire une fonction **hashCode(myString)** qui renvoie un entier. Il serait judicieux que sur 2 String différentes, la fonction **hashCode** renvoie 2 entiers différents (le plus souvent possible)

Principe : on va stocker la chaîne s dans mySet, à l'endroit où l'on aurait stocké l'entier hashCode(s)

- Ecrire une fonction **tab2StringSet(stringTab, taille)** qui renvoie un tel mySet
- Ecrire une fonction **stringContain(mySet, myString)** qui indique si myString appartient à mySet

Bonus : Puisque Python le permet, ajouter la fonction **hashCode** aux paramètres des fonctions **tab2StringSet** et **stringContain** (et/ou utilisez les lambda expressions)

On s'intéresse enfin au paramètre taille des fonctions **tab2Set**.

Pour que la fonction **contain** s'exécute rapidement, il faut que les tableaux de **mySet** restent relativement petits. Sur un texte littéraire quelconque (quelconque mais assez gros, par exemple [https://fr.wikisource.org/wiki/Quatrevingt-treize/I,\\_1](https://fr.wikisource.org/wiki/Quatrevingt-treize/I,_1)), déterminer la valeur du paramètre **taille** qui permet d'avoir des **mySet[i]** de taille inférieure à 100. Pour la fonction **hashCode**, on pourra opter pour le hashcode des String en Java (<https://docs.oracle.com/javase/1.5.0/docs/api/java/lang/String.html#hashCode%28%29>)

Autre source de données pour récupérer un gros tableau de string (1 million de mots environ) :

```
import urllib.request
with urllib.request.urlopen('http://norvig.com/big.txt') as
response:
    html = response.read().decode('utf-8')
tab=html.lower().split()
```

Fonctions Python utiles :

1. char to ascii : `ord('A')` vaut 65
2. Découper une String en token. : avec `split` du module `re`  
`import re`

## 8. Récursivité

### Exemple et rappel terminologique

```

FONCTION g (ENT n) RENVOIE ENT
DEBUT
    SI n<3
        ALORS RENVOIE 0
    FSI
    RENVOIE g(n-1)-1
FIN g

```

Calculer les valeurs de :

- $g(5)$
- $g(7)$
- $g(n)$ , pour un  $n$  quelconque

### Exercice préliminaire : Itération en récursivité

Contrainte : interdit d'utiliser des structures itératives (POUR ou ITER). Indiquer pour chaque fonction, l'algorithme qui comprend la première instruction d'appel.

- a. Ecrire une fonction *IterRec* récursive qui prend en paramètre un entier  $n$  et qui produit l'affichage de  $n$  fois le message « bonjour ».
- b. Ecrire une fonction *parTab* récursive qui prend en paramètre un tableau d'entiers et qui en affiche les valeurs de gauche à droite. Comment modifier l'ordre d'affichage des éléments sans changer le premier appel de la fonction *parTab* ?

def parTab(tab, indice):

parTab([1,2,3,4],0)

- c. Ecrire une fonction *SomRec* récursive qui prend en paramètre un tableau d'entiers et qui renvoie la somme de ses éléments.

### Exercice préliminaire : suite de Fibonacci

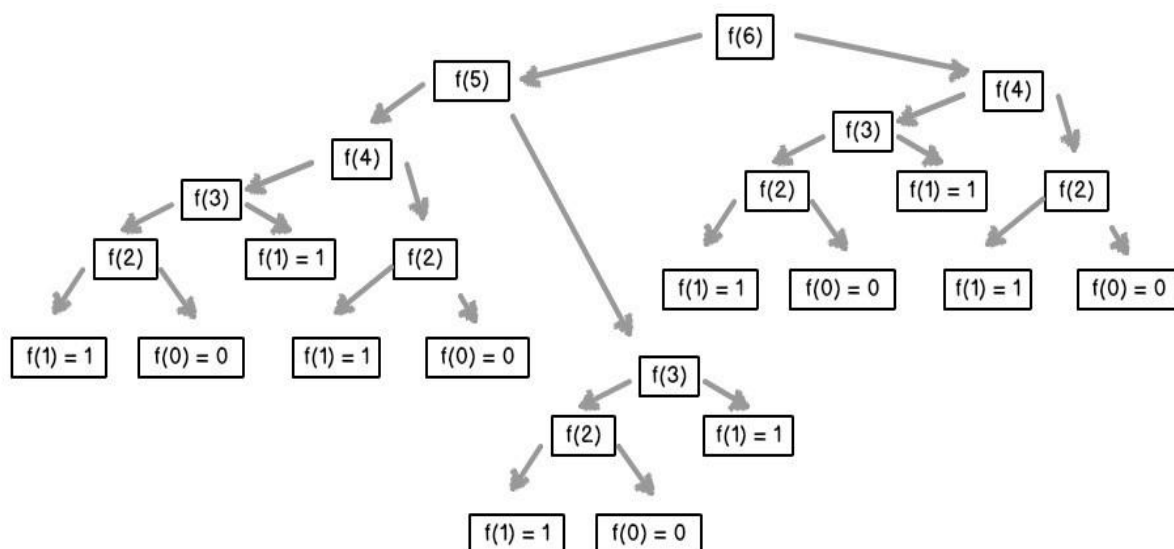
- a. Ecrire une fonction récursive *fibo*(int  $n$ ) qui renvoie la valeur du  $n$ -ième terme de la suite de Fibonacci définie par :

$$U_n = U_{n-1} + U_{n-2} \quad \forall n > 1 \text{ avec } U_0 = 0 \text{ et } U_1 = 1$$

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7  | 8  | 9  | 10 | 11 | 12  | 13  |
|---|---|---|---|---|---|---|----|----|----|----|----|-----|-----|
| 0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 | 89 | 144 | 233 |

- b. Ecrire une fonction itérative équivalente  
 c. Comparer le temps d'exécution pour  $n = 25$ ,  $n=35$  et  $n=45$

```
import time
start = time.time()
for i in range (100000): x=x**i
end = time.time()
print(end - start)
```



### Exercice 1 : affichages des chaines de bits de taille n

Par exemple, pour  $n = 2$ , la console doit afficher : 00, 01, 10, 11

Soit `tab` un tableau de taille  $n$ . On cherche à remplir ce tableau de toutes les manières possibles avec des 0 et des 1. Pour ce faire, on va écrire une fonction récursive `void chainesBits(int [] tab, int ind)` qui devra :

- affecter une valeur à `tab[ind]` (0 ou 1)
- s'appeler sur l'indice d'après

Début de solution à compléter :

```
def chainesBits ( tab, ind) :
    if ???
    else :
        tab[ind]=0
        chainesBits (tab, ind+1)
        tab[ind]=1
        chainesBits (tab, ind+1)

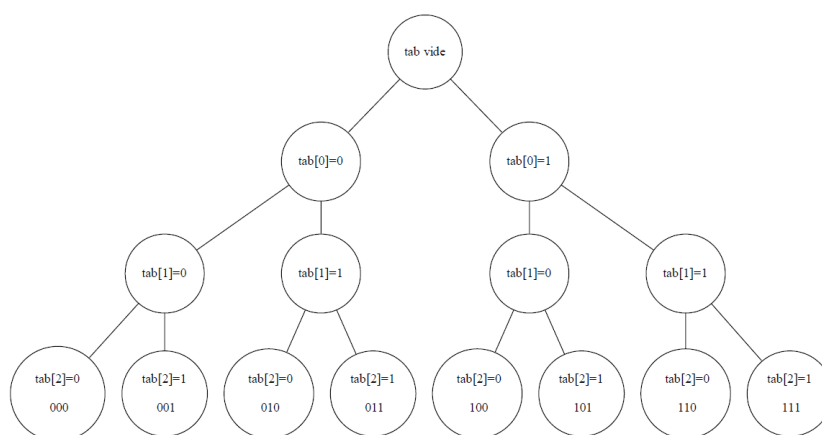
def printChainesBits(n) :
    tab=[0]*n
    chainesBits(tab,0)
```

Correction :

|   |   |
|---|---|
|   | def chainesBits ( tab, ind) :           |
| C | if ind == len(tab):print(tab)<br>else : |
| A | tab[ind]=0<br>chainesBits (tab, ind+1)  |
| B | tab[ind]=1<br>chainesBits (tab, ind+1)  |

Notation : chainesBits ( tab, ind) = cb(ind)

|              |              |              |       |         |
|--------------|--------------|--------------|-------|---------|
| cb(0)        |              |              |       | Console |
| A : tab[0]=0 | cb(1)        |              |       |         |
|              | A : tab[1]=0 | cb(2)        |       |         |
|              |              | A : tab[2]=0 | cb(3) |         |
|              |              |              | C →   | 000     |
|              |              | B : tab[2]=1 | cb(3) |         |
|              |              |              | C →   | 001     |
|              | B : tab[1]=1 | cb(2)        |       |         |
|              |              | A : tab[2]=0 | cb(3) |         |
|              |              |              | C →   | 010     |
|              |              | B : tab[2]=1 | cb(3) |         |
|              |              |              | C →   | 011     |
| B : tab[0]=1 | cb(1)        |              |       |         |
|              | A : tab[1]=0 | cb(2)        |       |         |
|              |              | A : tab[2]=0 | cb(3) |         |
|              |              |              | C →   | 100     |
|              |              | B : tab[2]=1 | cb(3) |         |
|              |              |              | C →   | 101     |
|              | B : tab[1]=1 | cb(2)        |       |         |
|              |              | A : tab[2]=0 | cb(3) |         |
|              |              |              | C →   | 110     |
|              |              | B : tab[2]=1 | cb(3) |         |
|              |              |              | C →   | 111     |

**Bonus :**

Modifier votre fonction de manière à stocker les chaînes obtenues dans une liste listeRes

```
void chainesBits2 (int [] tab, int ind, listeRes),
```

où listeRes est une liste des (different états du) tableaux tab

Attention, cette question nécessite d'utiliser les passage par reference des parameters de la fonction. (c'est plus compliqué via la valeur de retour) . Vous aller sans doute avoir besoin de stocker des copies du tableau tab. En python, liste[:] est une copie de liste (donc de reference différente)

**Exercice 2 [masque booléen] : Sous séquence de somme nulle**

Ecrire une fonction sousSeqSomNulle(liste, ind, tab) qui prend en paramètre un tableau d'entiers, et qui indique si la somme de certaines de ses valeurs (en nombre quelconque et non nécessairement côte à côte) vaut 0. Par exemple, sur le tableau tab = {4 , 6, 5, -3, 1, 9, -12, 40} la fonction doit afficher (textuellement) « 6+(-3)+9+(-12) = 0 ».

Indice : Utiliser un principe de masque booléen

|   |   |   |    |   |   |     |    |
|---|---|---|----|---|---|-----|----|
| 4 | 6 | 5 | -3 | 1 | 9 | -12 | 40 |
| 0 | 1 | 0 | 1  | 0 | 1 | 1   | 0  |

Pour ce tableau, les masques suivants sont solution :

```
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 1, 1, 1, 1, 1, 0]
[0, 1, 0, 1, 0, 1, 1, 0]
[0, 1, 1, 0, 1, 0, 1, 0]
[1, 1, 1, 1, 0, 0, 1, 0]
```

### Exercice 2.5 [masque booléen] : Nombre de montants à partir d'un ensemble de pièces

Le problème est le suivant : On dispose d'un ensemble de  $n$  pièces. La pièce numéro  $i$  vaut  $\text{tab}[i]$  euros. On cherche à savoir combien de montants différents peut-on régler avec ces pièces sans avoir à rendre de monnaie.

par exemple, si  $\text{tab}=[1, 2, 5, 5]$ , on peut régler les montants suivants :

0, 1, 2, 3, 5, 6, 7, 8, 10, 11, 12 et 13. Il y a donc 12 montants

**Version 1** : en listant tous les montants payables sans retour de monnaie (quitte à les afficher plusieurs fois)

0, 5, 5, 10, 2, 7, 7, 12, 1, 6, 6, 11, 3, 8, 8, 13

**Version 2** : en affichant la liste des montants payables

{0, 1, 2, 3, 5, 6, 7, 8, 10, 11, 12, 13}

**indice** : ajouter comme paramètre un set et y accumuler les montants

### Exercice 3 énumération des n-uplets

En s'inspirant de la fonction qui affiche les chaînes de bits de taille  $n$ , écrire une fonction qui affiche tous les chaînes (liste) de taille  $n$  dont les valeurs sont comprises entre 0 (inclus) et un entier  $\text{max}$  (exclu)

par exemple, si  $n=2$  et  $\text{max} = 3$ , votre fonction doit afficher :

[0, 0]  
[0, 1]  
[0, 2]  
[1, 0]  
[1, 1]  
[1, 2]  
[2, 0]  
[2, 1]  
[2, 2]

### Exercice 4 permutations

Adapter votre fonction de manière à afficher toutes les permutations de taille  $n$ , c'est-à-dire les liste formées d'entiers compris entre 0 (inclus) et  $n$  (exclu) sans valeurs répétées

Par exemple, pour  $n = 3$ , votre fonction doit afficher :

[0, 1, 2]  
[0, 2, 1]  
[1, 0, 2]  
[1, 2, 0]  
[2, 0, 1]  
[2, 1, 0]



### Exercice 5 [chaîne bits & backtracking] : Problème des mots bien parenthésés de taille n

On considère les mots formés uniquement de parenthèses. Par exemple  $w = '())()'$ .

1. Ecrire une fonction `bienBalance` qui prend en paramètre un tel mot (un tableau de char plutôt qu'une string)  
et qui renvoie `true` si les parenthèses sont bien balancées (Par exemple  $w = '(()())'$ ) et `false` sinon

| bien balancées        | mal balancées        |
|-----------------------|----------------------|
| <code>((()))</code>   | <code>)()()()</code> |
| <code>((())())</code> | <code>((()</code>    |
| <code>()</code>       | <code>)()</code>     |

2. Ecrire une fonction `printAll(n)` qui affiche tous les mots de parenthèses bien balancées de taille n
3. idem avec un mécanisme de backtracking

pour  $n = 6$  :

```
(( ( ( ) ) )
( ( ) ( ) )
( ( ) ) ( )
( ) ( ( ) )
( ) ( ) ( )
```

**Pour passer du tableau de char à une string :**

avec `tab = ['(', '(', '(', '(', '(', '(']`

```
' '.join(tab) vaut '(' '(' '(' '(' '(' '('
```

**Pour passer d'une string à un tableau de char :**

```
list('(') () () () () () ')
```

### Exercice 5 [nuplets & backtracking] : Problème des n-reines

Le but dans le problème des n reines est de placer n dames d'un jeu d'échecs sur un échiquier de  $n \times n$  cases sans que les dames ne puissent se menacer mutuellement, conformément aux règles du jeu d'échecs (la couleur des pièces étant ignorée). Par conséquent, deux dames ne devraient jamais partager la même rangée, colonne, ou diagonale. L'objectif est de trouver une solution en fonction d'un entier n fourni par l'utilisateur.

On propose coder une configuration des n reines sur l'échiquier à l'aide d'un tableau mono indice. Par exemple, pour  $n = 4$ , une des solutions est la configuration suivante :

|   |   |   |   |
|---|---|---|---|
|   |   | x |   |
| x |   |   |   |
|   |   |   | x |
|   | x |   |   |

Codage par numérotation des reines dans chaque colonne (vers le bas en partant de 0) :

|   |   |   |   |
|---|---|---|---|
| 1 | 3 | 0 | 2 |
|---|---|---|---|

Tableaux de test :

[1, 3, 0, 2] pour  $n = 4$

[0, 4, 7, 5, 2, 6, 1, 3] pour  $n = 8$

- version 1 : en modifiant la fonction nuplets de manière à parcourir tous les tableaux de n entiers strictement inférieurs à n.
- version 2 : **backtracking** : en incorporant un mécanisme de retour sur trace qui consiste à revenir légèrement en arrière sur des décisions prises afin de sortir d'un blocage. Par exemple, ceci permet de ne pas continuer à « avancer » si le début du tableau est

|   |   |  |  |
|---|---|--|--|
| 1 | 1 |  |  |
|---|---|--|--|

- On pourra définir une fonction *inserable*(ENT[] tab, ENT val, ENT r) qui renvoie un booléen valant VRAI si la valeur val est insérable dans le tableau tab au rang r, et FAUX sinon. Par exemple, *inserable*(tab, 1, 1) doit renvoyer FAUX si  $\text{tab} = \{1, -, -, -\}$ . («-» désigne des valeurs pas encore attribuées. On peut le remplacer par -1)

|   |  |  |  |
|---|--|--|--|
| 1 |  |  |  |
|---|--|--|--|

Par exemple, pour la 2<sup>ème</sup> case du tableau, les valeurs 0, 1, et 2 ne sont pas insérables

- On pourra définir une fonction *nReines*(ENT[] tab, ENT rang) qui renvoie un booléen valant VRAI si il existe une solution, et FAUX sinon
- Modifier votre algo de manière à lister toutes les configurations gagnantes pour un n donné

**Bonus (hors récursivité):**

Traiter le même problème (backtracking inclus) en générant le code.

Utiliser la fonction `eval('mon code')` qui exécute *mon code*

cf <https://docs.python.org/3/library/functions.html>

## 9. Gain de performance : méthodes générales et ad-hoc

Dans les problèmes ci dessous, vous devez fournir une solution brute-force et une solution plus rapide.

Pour chaque exercice, vous devez

- Déterminer la complexité des 2 solutions
- Evaluer la performance de vos deux solutions, et déterminer le gain de performance (en % moyen) de la solution rapide.

### Exercice 1 Recherche de la plus grande sous chaine commune

1. Ecrire une fonction *souchmax* qui prend en paramètre 2 chaines de caractère *adn1* et *adn2*, et qui renvoie une chaine qui vaut la plus grande sous chaine communes à *adn1* et *adn2*. Par exemple, *souchmax*("crotale", "hippopotame") doit renvoyer "ota". Si il existe plusieurs sous chaines communes, renvoyer celle qui apparait en premier dans *adn1*.  
Remarque : *adn1* et *adn2* pourront être des tableaux de caractères, ou bien même des tableaux d'entiers.

2. Solution rapide

Indice : de baser sur le tableau à double indice <sup>1</sup>suivant (à construire à partir des 2 chaines) :

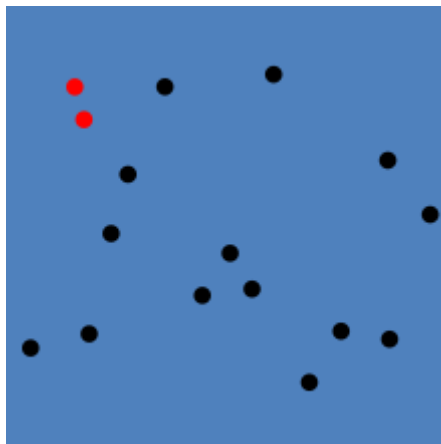
|   | H | I | P | P | O | P | O | T | A | M | E |
|---|---|---|---|---|---|---|---|---|---|---|---|
| C |   |   |   |   |   |   |   |   |   |   |   |
| R |   |   |   |   |   |   |   |   |   |   |   |
| O |   |   |   |   | 1 |   | 1 |   |   |   |   |
| T |   |   |   |   |   |   |   | 2 |   |   |   |
| A |   |   |   |   |   |   |   |   | 3 |   |   |
| L |   |   |   |   |   |   |   |   |   |   |   |
| E |   |   |   |   |   |   |   |   |   |   | 1 |

---

<sup>1</sup> grille = `[[0]*n for i in range(m)]`

## Exercice 2 Paire de points la plus proche

On considère un ensemble de  $n$  points du plan et on cherche à déterminer la paire de points la plus proche. Ce problème a des applications en contrôle aérien ou maritime.

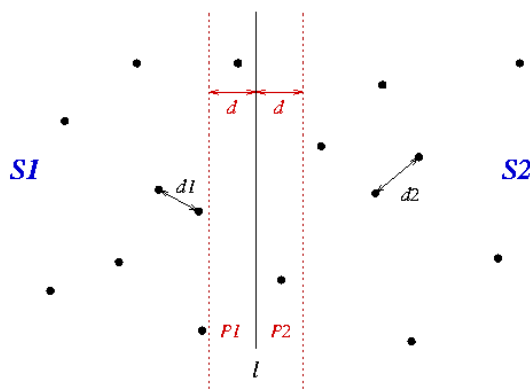


Les coordonnées de  $n$  points peuvent être rassemblées dans 2 tableaux  $x$  et  $y$  de taille  $n$

Par exemple, pour  $n = 3$

les points  $M_0=(2, 6)$ ,  $M_1=(4,3)$ ,  $M_2=(5,1)$ , sont codés dans les tableaux  $x=\{2,4,5\}$  et  $y=\{6, 3, 1\}$

- ➔ Ecrire une fonction float `dist(int a, int b, int c, int d)` qui renvoie la distance euclidienne entre les points de coordonnées  $(a, b)$  et  $(c, d)$ .
- ➔ Ecrire une fonction void `ppp(int[] x, int[] y)` qui affiche la distance entre la paire de points la plus proche.
- ➔ version rapide : diviser pour régner



### Pour visualiser le nuage de points en python :

Après l'installation du package matplotlib,

```
import matplotlib.pyplot as plt
plt.scatter(x, y)
plt.show()
```

### Exercice 3 Sous tableau de somme maximale

Le but de l'exercice est de déterminer pour un tableau d'entiers quelconques, le sous tableau dont la somme des éléments est la plus grande.

- a. Ecrire une fonction *sommesstab* qui prend en paramètre un tableau d'entier *tab*, et 2 entiers *i* et *j*, et qui renvoie un entier valant la somme des cases du tableau *tab* entre les indices *i* et *j* (inclus).

Par exemple, si  $\text{tab} = \{-6, 12, -7, 0, 14, -7, 5\}$ ,

|    |    |    |   |    |    |   |
|----|----|----|---|----|----|---|
| 0  | 1  | 2  | 3 | 4  | 5  | 6 |
| -6 | 12 | -7 | 0 | 14 | -7 | 5 |

$\text{sommesstab}(\text{tab}, 1, 4) = \text{tab}[1] + \text{tab}[2] + \text{tab}[3] + \text{tab}[4] = 12 - 7 + 0 + 14 = 19$

- b. Ecrire une fonction qui prend en paramètre un tableau d'entiers (positifs, nuls, ou négatifs), et renvoie la somme des éléments du sous tableau qui la rend maximale. On ne s'intéressera ici qu'aux sous tableaux correspondant à des cases *adjacentes* du tableau initial (il ne peut pas y avoir de trou).

Par exemple, si  $T = \{-6, 12, -7, 0, 14, -7, 5\}$  alors le sous tableau à déterminer est  $T' = \{12, -7, 0, 14\}$  car 19 est la somme partielle de cases adjacentes la plus grande (par exemple, la somme du sous tableau  $T'' = \{-6, 12, -7\}$  vaut 1, ce qui est moins que 19).

- Que vaut le sous tableau si le tableau ne contient que des entiers positifs ?
  - Que vaut le sous tableau si le tableau ne contient que des entiers négatifs ?
- c. Donner le nombre d'itérations de votre algorithme dans le pire des cas en fonction de la taille *n* du tableau initial
- d. Tenter de réduire ce nombre en raisonnant par récurrence. (\*\*)