
Informatique en CPGE

El Hadiq Zouhair

Feb 19, 2023

CONTENTS

1 Arbres binaires	3
1.1 Définition formelle d'un arbre binaire	3
2 Arbres Binaires de Recherche	7
2.1 Insertion et suppression	9
2.2 Requêtes dans un arbre binaire de recherche	11
3 Programmation dynamique	15
3.1 Introduction	15
3.2 Où est le problème ?	16
3.3 Une solution proposée par la programmation dynamique de type (Bottom UP)	16
3.4 Une solution proposée par la programmation dynamique de type (Top Down)	18
4 Programmation Dynamique et optimisation	21
4.1 Démarche de résoluton	21
4.2 Exemple: Problème Sac à Dos.	21
5 Les algorithmes gloutons	29
5.1 Introduction	29
5.2 Définition: Algorithme Heuristiques :	29
5.3 Définition: Algorithme glouton	29
6 TRAITEMENT DES IMAGES	31
6.1 Caractéristiques d'une image	31
6.2 Traitements basiques de l'image	38
6.3 Application de filtres	50
7 Algorithme KNN: Cours	57
7.1 Classification supervisée	57
7.2 La notion de distance:	58
7.3 Principe de l'algorithme	58
7.4 Mise en place de chaque étape	58
8 Tp Iris	61
8.1 Question 1	63
Bibliography	65

This is a small sample book to give you a feel for how book content is structured. It shows off a few of the major file types, as well as some sample content. It does not go in-depth into any particular topic - check out [the Jupyter Book documentation](#) for more information.

Check out the content pages bundled with this sample book to see more.

Table des matieres

- *Arbres binaires*
- *Arbres Binaires de Recherche*
- *Programmation dynamique*
- *Les algorithmes gloutons*
- *TRAITEMENT DES IMAGES*
- *Algorithme KNN: Cours*
- *Tp Iris*

bibliography

ARBRES BINAIRES

1.1 Définition formelle d'un arbre binaire

On appelle arité d'un nœud le nombre de branches qui en partent 3 . Dans la suite de notre cours, nous nous intéresserons plus particulièrement aux arbres binaires, c'est à dire ceux dont chaque nœud a au plus deux fils. Pour ne pas avoir à distinguer les nœuds suivant leur arité, il est pratique d'ajouter à l'ensemble des arbres binaires un arbre particulier appelé l'arbre vide. Ceci conduit à adopter la définition qui suit.

x est l'étiquette de la racine de A ; quant à Fg et Fd , ils sont appelés respectivement le sous-arbre gauche et le sous-arbre droit de l'arbre binaire A. De manière usuelle, on convient de ne pas faire figurer l'arbre vide dans les représentations graphiques des arbres binaires. Ainsi, suivant la représentation choisie les feuilles pourront désigner exclusivement l'arbre vide (et dans ce cas tous les nœuds seront d'arité égale à 2) ou alors les nœuds dont les deux fils sont vides (dans ce cas l'arité d'un nœud pourra être égale à 0, 1 ou 2). C'est cette seconde convention qui sera utilisée dans la suite de ce cours.

Définition. La taille(A) d'un arbre A est définie inductivement par les relations :

$\text{taille}(\emptyset) = 0$; Si $A = (F_g, x, F_d)$ alors $\text{taille}(A) = 1 + \text{taille}(F_g) + \text{taille}(F_d)$.

Définition. La hauteur: $h(A)$ d'un arbre A se définit inductivement par les relations :

$h(\emptyset) = 0$; Si $A = (F_g, x, F_d)$ alors $h(A) = 1 + \max(h(F_g), h(F_d))$.

Avec ces conventions, $\text{taille}(A)$ est le nombre de nœuds d'un arbre et $h(A)$ la longueur maximale du chemin reliant la racine à une feuille, autrement dit la profondeur maximale d'un nœud. La définition Python de ces fonctions est immédiate :

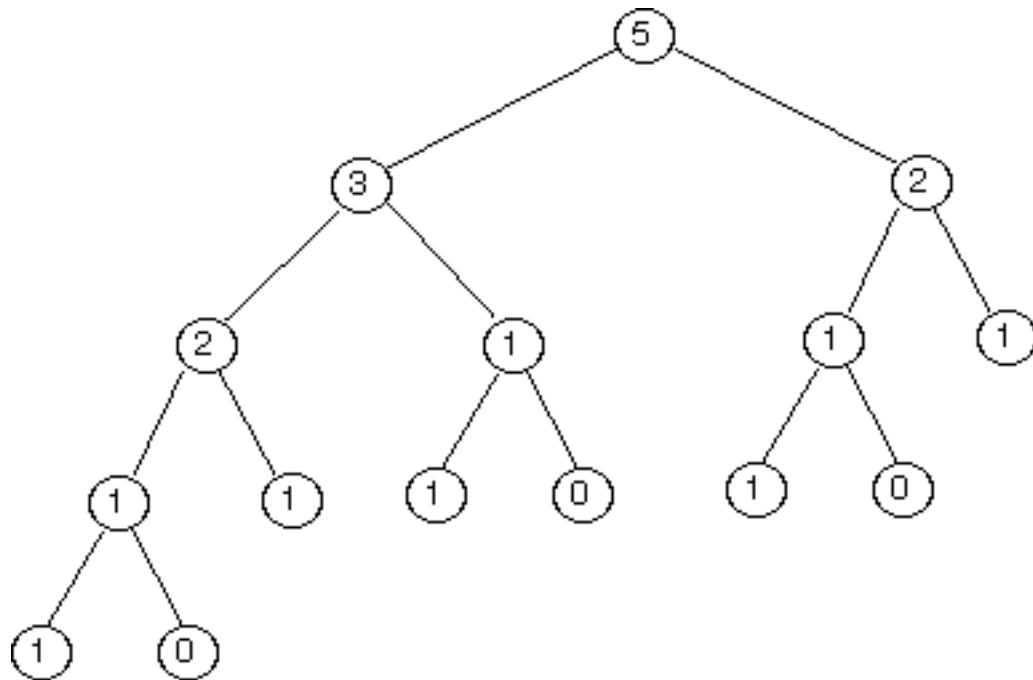
```
Vide=[]
f1=[1,Vide,Vide]
f0=[0,[],[]]
```

```
f2=[1,f1,f0]
```

```
f2
```

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

```
def FiboA(n):
    if n<=1:
        return [n,[],[]]
    Fg=FiboA(n-1)
    Fd=FiboA(n-2)
    racine=Fg[0]+Fd[0]
    return [racine,Fg,Fd]
```



F4=FiboA(4)

```

def taille(A):
    if A==[]:
        return 0
    return 1+taille(A[1])+taille(A[2])
    
```

```

def hauteur(A):
    if A==[]:
        return 0
    return 1+max(hauteur(A[1]),hauteur(A[2]))
    
```

hauteur(F4)

4

```

def parcour_prefixe(A):
    if A==[]:
        return []
    return [A[0]]+parcour_prefixe(A[1])+parcour_prefixe(A[2])
    
```

```

def parcour_postfixe(A):
    if A==[]:
        return []
    return parcour_postfixe(A[1])+parcour_postfixe(A[2])+[A[0]]
    
```

parcour_prefixe(F4)

```
[3, 2, 1, 1, 0, 1, 1, 1, 0]
```

```
def parcour_infixe(A):
    if A==[]:
        return []
    return parcour_infixe(A[1])+[A[0]]+parcour_infixe(A[2])
```

```
print("prefixe: ",parcour_prefixe(F4))
print("postfixe: ",parcour_postfixe(F4))
print("infixe: ",parcour_infixe(F4))
```

```
prefixe: [3, 2, 1, 1, 0, 1, 1, 1, 0]
postfixe: [1, 0, 1, 1, 2, 1, 0, 1, 3]
infixe: [1, 1, 0, 2, 1, 3, 1, 1, 0]
```

```
def parcour_en_largeur(A):
    L1,L2=[A],[]
    larg=[]
    while L1!=[]:
        for E in L1:
            if E!=[]:
                larg.append(E[0])
                L2.extend([E[1],E[2]])
    L1=L2
    L2=[]
    return larg
```

```
parcour_en_largeur(FiboA(4))
```

```
[3, 2, 1, 1, 1, 0, 1, 0]
```

```
print("La taille de F(4) est {}, et son hauteur est {}".format(taille(F4),
    hauteur(F4)))
```

```
La taille de F(4) est 9, et son hauteur est 4
```


ARBRES BINAIRES DE RECHERCHE

Indication: Concernant les cellules qui contiennent des théorèmes enlever le commentaire (`<!-->`) pour visualiser la preuve apres avoir essayer le prouvé vous même

```
import numpy as np
import random
```

Un arbre binaire de recherche [Jean-Pierre-Becirspahic., n.d.] (en abrégé : ABR) permet l'implémentation sous forme d'arbre binaire de certaines structures de données stockant des éléments formés d'une clé et d'une valeur, tels les dictionnaires 7. Nous allons donc considérer un ensemble ordonné de clés C ainsi qu'un ensemble de valeurs V , et utiliser des arbres binaires étiquetés par $E = C \times V$. Les arbres binaires de recherche supportent nombre d'opérations qu'on utilise dans les structures de données, en particulier :

- la recherche d'un valeur associée à une clé donnée ;
- la recherche de la valeur associée à la clé maximale (ou minimale) ;
- la recherche du successeur d'une clé c , c'est à dire la valeur associée à la plus petite des clés strictement supérieures à c ;
- la recherche du prédécesseur d'une clé ;
- et bien sûr l'insertion ou la suppression d'un nouveau couple clé/valeur.

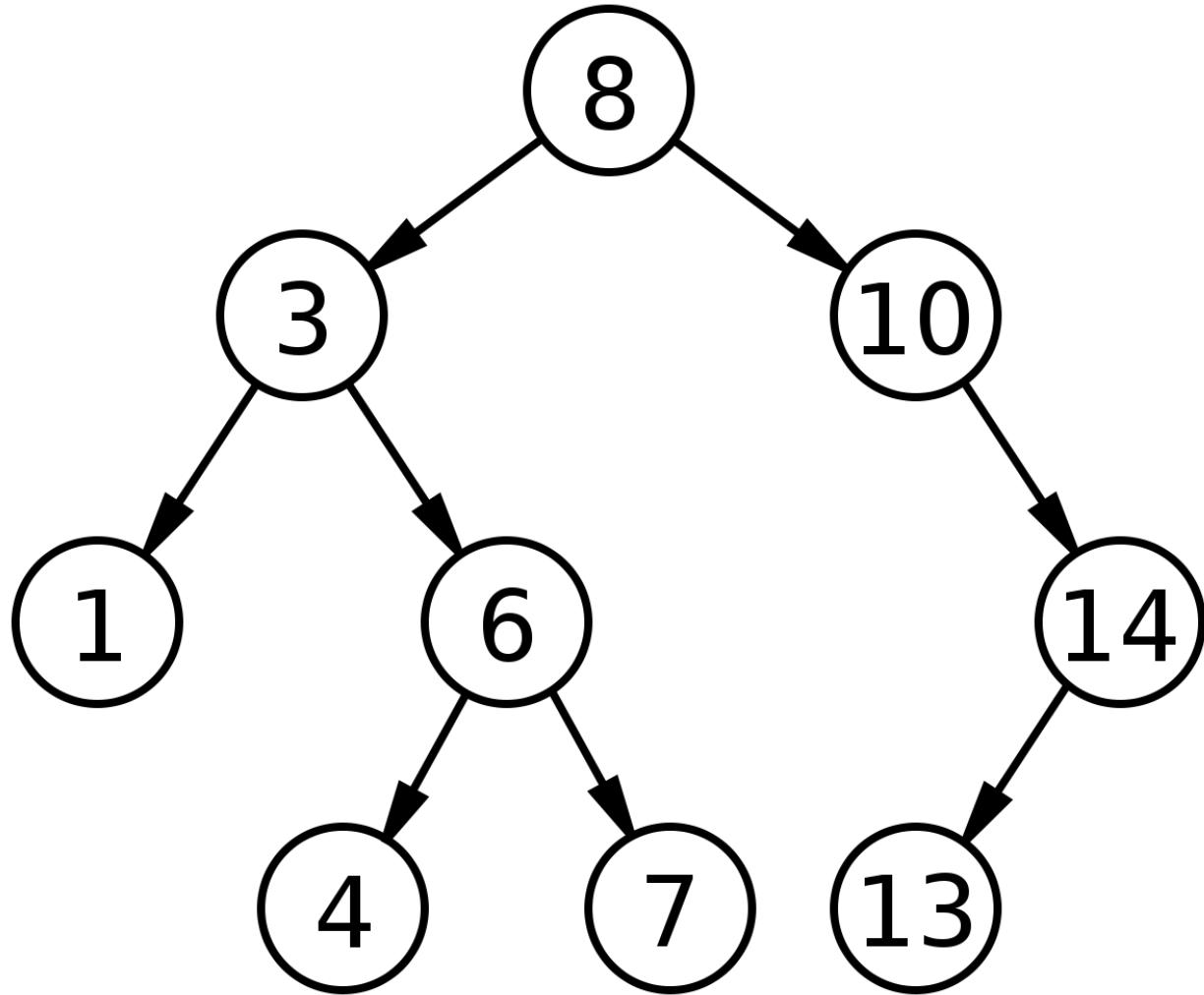


Figure 1 – Un exemple d’arbre binaire de recherche (seules les clés ont été représentées)

Définition. un arbre binaire A est un arbre binaire de recherche s’il est vide ou égal à $[(c, v), Fg, Fd]$ où :

- Fg et Fd sont des arbres binaires de recherche ;
- toute clé de Fg est inférieure (ou égale) à c ;
- toute clé de Fd est supérieure (ou égale) à c .

Autrement dit, A est un arbre binaire de recherche lorsque tout noeud de A est associé à une clé supérieure ou égale à toute clé de son fils gauche, et inférieure ou égale à toute clé de son fils droit.

Théorème. Lors du parcours infixé d’un arbre binaire de recherche, les clés sont parcourues par ordre croissant.

Preuve. On procède par induction :

- si $A = []$, il n’y a rien à prouver ;
- si $A = [x, Fg, Fd]$, on suppose que les parcours infixes de Fg et de Fd se font par ordre de clés croissantes. Sachant que toute clé de Fg est inférieure à la clé de x et toute clé de Fd supérieure à cette dernière, le parcours $Fg \rightarrow x \rightarrow Fd$ est bien effectué par ordre croissant de clé. \rightarrow

2.1 Insertion et suppression

2.1.1 1. Insertion et suppression dans un arbre binaire de recherche quelconque

Insertion

Pour insérer le couple $(k; u) \in C \times V$ dans l'arbre A on procède ainsi :

- si $A = []$, on retourne l'arbre $((k, u), [], [])$;
- si $A = [(c, v), Fg, Fd]$ alors :
 - si $c = k$ on remplace le couple (c, v) par (k, u) et on insère (c, v) dans Fg ou Fd ;
 - si $c > k$ on insère (k, u) dans Fg ;
 - si $c < k$ on insère (k, u) dans Fd

```
def insertion_feuille(A, paire):
    if A==[]:
        A.extend([paire, [], []])
    elif paire[0]<=A[0][0]:
        insertion_feuille(A[1], paire)
    else:
        insertion_feuille(A[2], paire)
```

```
def remplire(A, L):
    for paire in L:
        insertion_feuille(A, paire)
```

```
def initialiser():
    random.seed(0) #Pour avoir les même valeurs aléatoires
    L=[random.randint(0,20) for _ in range(8)]
    L=[(e, "v"+str(e)) for e in L]
    #print("-->".join([str(p) for p in L]))
    A=[] #Initialisation par liste vide.
    remplire(A, L)
    #print(L)
    return A, L
```

```
A,L=initialiser()
print(A)
```

```
[(12, 'v12'), [(1, 'v1'), [], [(8, 'v8'), [], [(12, 'v12'), [(9, 'v9'), [], []], [(), ()]]], [(13, 'v13'), [], [(16, 'v16'), [(15, 'v15'), [], []], [()]]]]]
```

Cette fonction d'insertion à un coût en $O(h(A))$.

2.1.2 Suppression

Suppression de la valeur minimale

Nous aurons besoin d'une fonction qui prend en argument un arbre A et retourne le couple $(m; A')$ formé d'un élément m de clé minimale et de l'arbre $A' = A \setminus \{m\}$:

- Si $A = []$: Alors rien à supprimer
- Si $Fg = []$ alors supprimer la racine et remplacer l'arbre A par Fd .
- Sinon supprimer le couple minimale à partir de l'arbre gauche.

```
def supprime_min(A):
    if A==[]:
        return None
    if A[1]==[]:
        droped=A[0]
        A[:]=A[2]
    return droped
return supprime_min(A[1])
```

La suppression d'une clé k dans un arbre binaire de recherche consiste à supprimer le couple $(k; v)$ situé à la profondeur minimale dans l'arbre $A = [(c; v); Fg; Fd]$. On procède ainsi :

- si $k < c$, on supprime un élément de clé k dans Fg ;
- si $k > c$, on supprime un élément de clé k dans Fd ;
- si $k = c$, alors :
 - si $Fg = []$ on renvoie Fd ;
 - si $Fd = []$ on renvoie Fg ;
 - sinon, on supprime de Fd un élément m de clé minimale pour obtenir Fd' et on retourne $[m, Fg, Fd']$.

La suppression d'une clé k se réalise alors ainsi :

```
def supprimer(A,k):
    if A==[]:
        return None
    if k<A[0][0]:
        return supprimer(A[1],k)
    if k>A[0][0]:
        return supprimer(A[2],k)

    if A[1]==[]:
        dropped,A[:]=A[0],A[2]
    return dropped
    if A[2]==[]:
        dropped,A[:]=A[0],A[1]
    return dropped
    m=supprime_min(A[2])
    A[:]=[m,A[1],A[2]]
return A[0]
```

```
A,_=initialiser()
print(A)
```

```
[[(12, 'v12'), [(1, 'v1'), [], [(8, 'v8'), [], [(12, 'v12'), [(9, 'v9'), [], []], [(), ()]]], [(13, 'v13'), [], [(16, 'v16'), [(15, 'v15'), [], []], [()]]]]]
```

```
supprimer(A, 8)
print(A)
```

```
[[(12, 'v12'), [(1, 'v1'), [], [(12, 'v12'), [(9, 'v9'), [], []], [(), ()]]], [(13, 'v13'), [], [(16, 'v16'), [(15, 'v15'), [], []], [()]]]]]
```

La fonction de suppression a elle aussi un coût temporel en $O(h(A))$ puisque chaque appel récursif s'effectue sur un arbre de hauteur inférieur au précédent.

2.2 Requêtes dans un arbre binaire de recherche

2.2.1 Recherche d'une clé

Une des opérations les plus courantes dans un arbre binaire de recherche $A = [Fg; (c; v); Fd]$ est la recherche d'une valeur associée à une clé particulière k . La démarche est évidente :

- si $k = c$, retourner v ;
- si $k < c$, rechercher k dans Fg ;
- si $k > c$, rechercher k dans Fd .

Dans le cas où les clés ne sont pas toutes distinctes on retournera la valeur associée à la première clé égale à k rencontrée.

```
recherche=lambda A,k: None if A==[] else A[0][1] if A[0][0]==k else recherche(A[1], k) if k<A[0][0] else recherche(A[2], k)
```

```
recherche(A, 12)
```

```
'v12'
```

2.2.2 2.1 Recherche de la clé minimale / maximale

La recherche de la valeur associée à la clé minimale se poursuit dans le fils gauche tant que ce dernier n'est pas vide :

```
np.inf<10
```

```
False
```

```
def recherche_min(A):
    if A==[]:
        #return (+infinie,"no min") si l'arbre est vide
        return (np.inf,"no min")
    #sinon poursuive la recherche dans le sous arbre gauche
    min_g=recherche_min(A[1])
```

(continues on next page)

(continued from previous page)

```
if A[0][0]<min_g[0]:
    return A[0]
else:
    return min_g
```

```
A,_=initialiser()
recherche_min(A)
```

```
(1, 'v1')
```

La fonction retournant la valeur associée à la clé maximale est symétrique :

```
import numpy as np
type(np.infty)
```

```
float
```

```
def recherche_max(A):
    if A==[]:
        #return (-infinie,"no max") si l'arbre est vide
        return (-np.infty,"no max")
    #sinon poursivie la recherche dans le sous arbre droit
    max_d=recherche_max(A[2])
    if A[0][0]>=max_d[0]:
        return A[0]
    else:
        return max_d
```

```
recherche_max(A)
```

```
(16, 'v16')
```

2.2.3 Recherche du prédécesseur / successeur

$k \in C$ étant donné, il s'agit cette fois de retourner la valeur associée à la plus grande des clés c contenues dans l'arbre et vérifiant : $c < k$ (respectivement $c > k$)

```
def prédécesseur(A,k):
    if A==[]:
        return None
    if A[0][0]<k:
        pred=prédécesseur(A[2],k)
        return A[0][1] if pred == None else pred
    else:
        return prédécesseur(A[1],k)
```

```
print("prédécesseur de: ")
for e in L:
    print("\t{} < {}".format(prédécesseur(A,e[0]),e[1]))
```

```

prédécesseur de:
v9 < v12
v12 < v13
None < v1
v1 < v8
v15 < v16
v13 < v15
v9 < v12
v8 < v9

```

```

def successeur(A, k):
    if A==[]:
        return None
    if A[0][0]>k:
        succ=successeur(A[1],k)
        return A[0][1] if succ is None else succ
    else:
        return successeur(A[2],k)

```

```

print("prédécesseur | clé | successeur ")
for e in L:
    print("\t{} < {}<{}".format(prédécesseur(A,e[0]),e[1],successeur(A,e[0])))

```

```

prédécesseur | clé | successeur
v9 < v12<v13
v12 < v13<v15
None < v1<v8
v1 < v8<v9
v15 < v16<None
v13 < v15<v16
v9 < v12<v13
v8 < v9<v12

```

2.2.4 Complexité temporelle

Toutes ces fonctions ont à l'évidence un coût temporel en $O(h(A))$, ce qui explique tout l'intérêt qu'il peut y avoir à ce que l'arbre binaire de recherche soit équilibré : dans un arbre binaire quelconque d'ordre $n = \text{taille}(A)$, on peut affirmer que le coût d'une requête est un $O(n)$; dans le cas d'un arbre de recherche équilibré, on peut assurer un coût en $O(\log(n))$

PROGRAMMATION DYNAMIQUE

3.1 Introduction

```
from mesurer import mesure
import numpy as np
##cette fonction permet de mesurer le temps d'exécution d'une autre fonction
```

3.1.1 Un inconvénient de la programmation récursive

Nous allons nous intéresser au calcul du coefficient binomial $C_n^p = \binom{n}{p}$. Une solution consiste à utiliser la programmation récursive et la formule de Pascal, ce qui nous amène à écrire :

```
def binom(n, p):
    if p == 0 or n == p:
        return 1
    return binom(n- 1, p -1) + binom(n -1, p)
```

```
mesure(binom)(30,15)
```

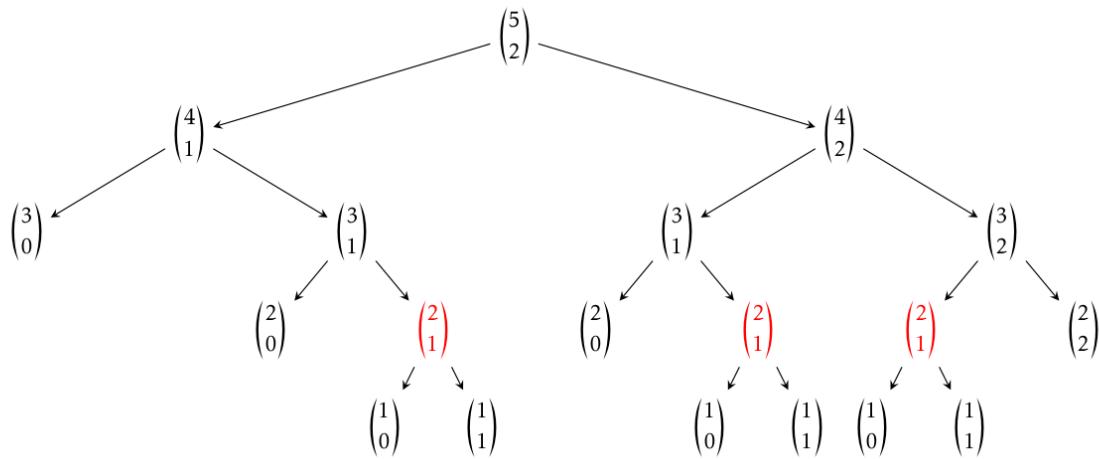
```
Temps d'execution :62.49358534812927 secondes
```

```
155117520
```

Malheureusement, cette fonction s'avère très peu efficace, même pour de relativement faibles valeurs de n et p : à titre d'illustration, il faut 42 secondes à mon ordinateur pour calculer $\text{binom}(30,15)$,

La raison en est facile à comprendre : lorsqu'on observe par exemple l'arbre de calcul de $\binom{5}{2}$ on constate qu'il ya des appels récursifs sont identiques et donc superflus (figure 1).

Nous pouvons par exemple constater que le calcul de $\binom{5}{2}$ nécessite de calculer trois $\binom{2}{1}$. Et l'expérience montre que le calcul de $\binom{30}{15}$, fait appel 40 116 600 au calcul de $\binom{2}{1}$.



3.2 Où est le problème ?

Le problème à résoudre, ici le calcul de $\binom{n}{p}$, se ramène à la résolution de deux sous-problèmes : le calcul de $\binom{n-1}{p-1}$ et de $\binom{n-1}{p}$, sous-problèmes qui sont en interaction.

Par exemple, on constate sur la figure 1 que le calcul de $\binom{4}{1}$ et le calcul de $\binom{4}{2}$ font tous deux appel au même sous-problème : le calcul de $\binom{3}{1}$.

Ainsi, la présence de sous-problèmes en interaction peut faire croître très rapidement la complexité d'une fonction, au point d'en rendre son usage rédhibitoire.

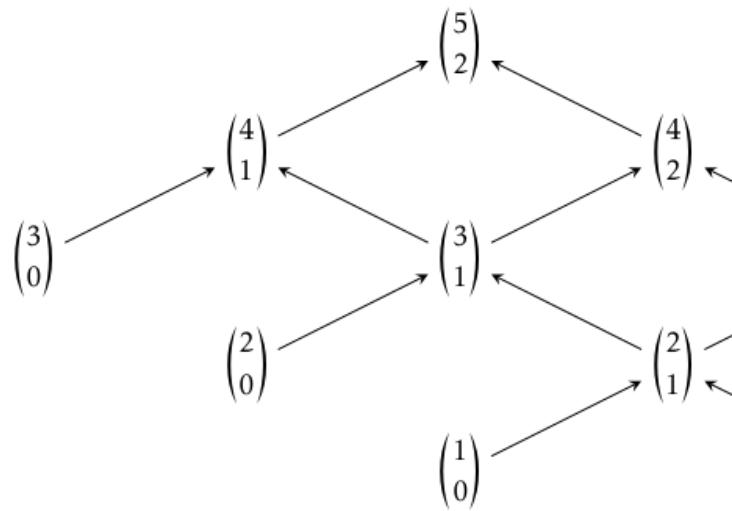
Tout comme les problèmes que l'on résout par une méthode « diviser pour régner », les problèmes que l'on résout par la programmation dynamique se ramènent à la résolution de sous-problèmes de tailles inférieures.

Mais à la différence de la méthode « diviser pour régner », ces sous-problèmes ne sont pas indépendants, ce qui impose d'accompagner la programmation récursive par une analyse fine des relations de dépendance, ou beaucoup plus simplement par l'utilisation de la mémoisation qui gère les relations de dépendance à notre place.

3.3 Une solution proposée par la programmation dynamique de type (Bottom UP)

La solution proposée par la programmation dynamique consiste à commencer par résoudre les plus petits des sous-problèmes, puis de combiner leurs solutions pour résoudre des sous-problèmes de plus en plus grands.

La solution proposée par la programmation dynamique consiste à commencer par résoudre les plus petits des sous-problèmes, puis de combiner leurs solutions pour résoudre des sous-problèmes de plus en plus grands.



Concrètement, le calcul de $\binom{5}{2}$ se réalise en suivant le schéma :

Pour réaliser ce type de solution on utilise souvent un tableau, ici un tableau bi-dimensionnel $(n + 1) \times (p + 1)$ (dont seule la partie pour laquelle $i > j$ sera utilisée).

Ce tableau sera progressivement rempli par les valeurs des coefficients binomiaux, en commençant par les plus petits:

	0	1	p	j
0	1				
1	1	1			
.	1		1		
.	1			1	
.	1				1
.	1				
.	1				
.	1				
n	1				$\binom{n}{p}$

A red arrow points from the value 1 at position (1,1) to the value 1 at position (2,2). To the right of the table, there is a diagram showing three stacked boxes. The top box contains $\binom{i-1}{j-1}$, the middle box contains $\binom{i-1}{j}$, and the bottom box contains $\binom{i}{j}$.

Il faut faire attention à bien respecter la relation de dépendance (modélisée par les flèches sur le schéma ci-dessus) pour remplir les cases de ce tableau : la case destinée à recevoir la valeur de $\binom{i}{j}$ ne peut être remplie qu'après les cases destinées à recevoir $\binom{i-1}{j-1}$ et $\binom{i-1}{j}$.

```
def binom_bottom_up(n, p):
    ## Algorithme itératif avec liste
    t = np.zeros((n + 1, p + 1), dtype=np.int64)
    for i in range(0, n + 1):
        t[i, 0] = 1
    for i in range(1, p + 1):
        t[i, i] = 1
    for i in range(2, n + 1):
        for j in range(1, min(p, i) + 1):
            t[i, j] = t[i - 1, j - 1] + t[i - 1, j]
    return t[n, p]
```

Au prix d'un coût spatial (la création du tableau) cet algorithme est bien plus efficient que l'algorithme récursif initial puisque sa complexité temporelle et spatiale est maintenant en $O(np)$. Remarque. Notons que cette solution n'est pas encore optimale : il est facile de constater sur le schéma de dépendance que l'algorithme ci-dessus remplit des cases inutiles pour le calcul de $\binom{n}{p}$: seules celles qui sont colorées sont nécessaires.

Un autre inconvénient, plus important celui-là réside dans la perte de lisibilité de l'algorithme, comparativement à l'algorithme récursif. L'idéal serait donc de combiner l'élégance de la programmation récursive avec l'efficacité de la programmation dynamique.

3.4 Une solution proposée par la programmation dynamique de type (Top Down)

La solution existe, elle porte le nom de mémoïsation.

Elle consiste à associer à la fonction un dictionnaire qui va mémoriser le résultat du calcul réalisé. Ainsi, à chaque fois que le programme aura besoin de calculer une valeur, il ira voir dans le dictionnaire si la valeur dont il a besoin a déjà été calculée, et ne réalisera le calcul que dans le cas contraire, en ajoutant ensuite la nouvelle valeur calculée au dictionnaire.

Le calcul du coefficient binomial va alors prendre la forme qui suit :

```
binom_dict = {}
def binom_top_down(n, p):
    if (n, p) not in binom_dict:
        if p == 0 or n == p:
            b = 1
        else:
            b = binom_top_down(n - 1, p - 1) + binom_top_down(n - 1, p)
        binom_dict[(n, p)] = b
    return binom_dict[(n, p)]
```

On peut observer que le programme récursif se retrouve presque mot pour mot lignes 5 à 8. ! Calculons $\binom{5}{2}$ avec cette fonction, puis observons le contenu du dictionnaire :

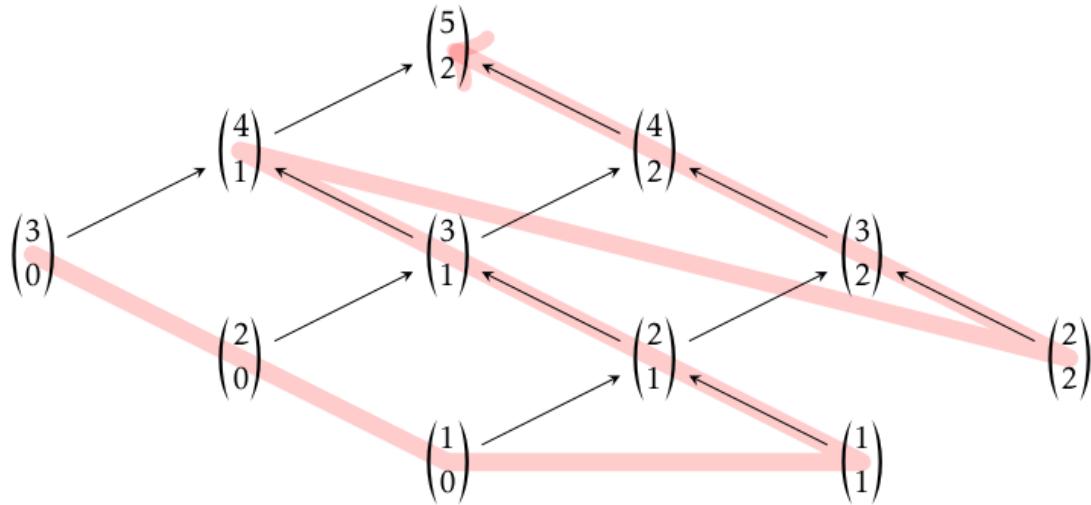
```
binom_top_down(5, 2)
```

10

```
print(binom_dict)
```

```
{(3, 0): 1, (2, 0): 1, (1, 0): 1, (1, 1): 1, (2, 1): 2, (3, 1): 3, (4, 1): 4, (2, 2): 1, (3, 2): 3, (4, 2): 6, (5, 2): 10}
```

On peut constater qu'on y retrouve les 10 valeurs nécessaires pour réaliser ce calcul. J'ai représenté figure suivante l'ordre dans lequel ces valeurs ont été introduites dans le dictionnaire.



PROGRAMMATION DYNAMIQUE ET OPTIMISATION

Un des principaux champs d'applications de la programmation dynamique est la résolution de problèmes d'optimisation. Il s'agit de problèmes dont chaque solution possède une valeur.

On cherche alors une solution de valeur optimale (minimale ou maximale). Il existe de nombreuses techniques pour résoudre ce genre de problèmes et la programmation dynamique en fait partie. Des méthodes gloutonnes peuvent également marcher.

Il est donc logique de se demander à quelles conditions doit-on utiliser la programmation dynamique pour résoudre un problème d'optimisation.

Il faut premièrement que l'ensemble des éléments constituant le problème soit discret et fini . Par exemple pour la recherche de plus courts chemins, ces éléments sont les sommets du graphe et les arêtes les reliant entre eux.

Ensuite, une solution optimale au problème global doit induire des

- solutions optimales aux sous-problèmes.
- Enfin, il est nécessaire que les sous-problèmes ne soient pas indépendants .

4.1 Démarche de résolution

On retrouvera généralement quatre étapes dans la conception d'une solution optimale en utilisant la technique de programmation dynamique :

- Caractériser la structure d'une solution optimale.
- Définir par récurrence la valeur d'une solution optimale.
- Calculer la valeur d'une solution optimale par une méthode Top Down ou Bottom Up.
- Construire la solution optimale .

4.2 Exemple: Problème Sac à Dos.

Le problème du Sac à Dos est un problème classique en informatique. Il modélise une situation analogue au remplissage d'un sac.

Une personne veut remplir un sac à dos ne pouvant pas supporter plus d'un certain poids $W_{max} \in \mathbb{N}$, et elle dispose de n objets On note l'ensemble des objets par $O = \{1, \dots, n\}$.

Chaque objet i a une valeur v_i et un poids w_i .

Le problème est de trouver un ensemble d'objets tels que:

- tous les objets de cet ensemble puissent être mis dans le sac.
- la somme des valeurs de ces objets soit maximale.

Le problème d'optimisation correspond à trouver un sous-ensemble I de O d'objets dont le poids total est inférieur à W et dont la valeur totale $\sum_{i \in I} v_i$ soit maximum.

```
#exemple
# La liste des poids
WL=[3,9,6,3]
#La liste des valeurs
VL=[4,2,10,9]
#le poids maximal
Wmax=15
#le nombre d'objets
n=len(WL)
```

```
S=list(zip(WL,VL))
```

```
print(S)
sorted(S,key=lambda e:(e[1]/e[0]),reverse=True)
```

```
[(3, 4), (9, 2), (6, 10), (3, 9)]
```

```
[(3, 9), (6, 10), (3, 4), (9, 2)]
```

```
def sacAdos_naive(VL,WL,Wmax):
    if len(VL)==0:
        return 0
    if Wmax==0:
        return 0
    v,w=VL[-1],WL[-1]
    if w<=Wmax:
        return max(v+sacAdos_naive(VL[:-1],WL[:-1],Wmax-w),sacAdos_naive(VL[:-1],WL[:-1],Wmax))
    return sacAdos_naive(VL[:-1],WL[:-1],Wmax)
```

```
def sacAdos_naive2(k,Wmax):
    #La même chose avec la précédent sauf on prend comme parametre le nombre d
    #éléments de la liste et non pas la liste
    if k==0:
        return 0
    if Wmax==0:
        return 0
    v,w=VL[k],WL[k]
    if w<=Wmax:
        return max(v+sacAdos_naive2(k-1,Wmax-w),sacAdos_naive2(k-1,Wmax))
    return sacAdos_naive2(k-1,Wmax)
```

```
print("la valeur maximal par la méthode naïve est ",sacAdos_naive(VL,WL,Wmax))
print("la valeur maximal par la méthode naïve est ",sacAdos_naive2(n-1,Wmax))
```

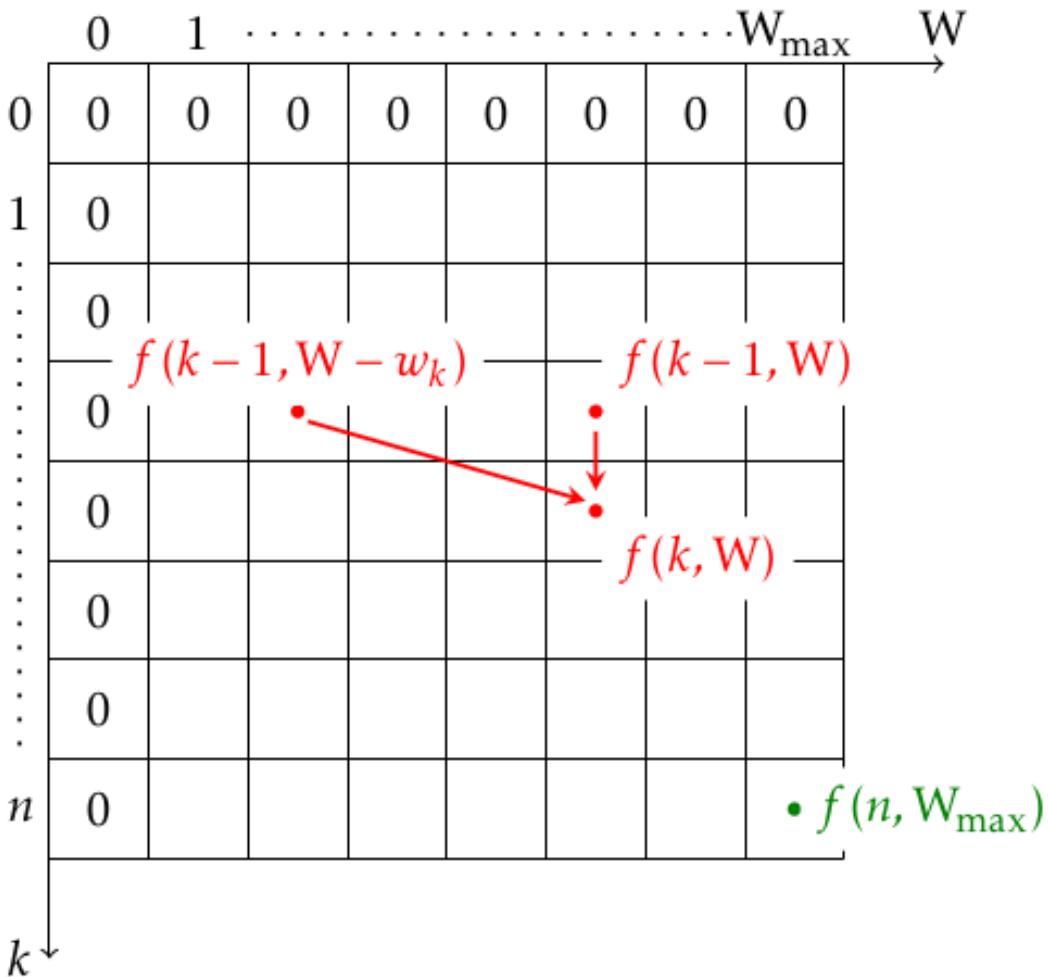
```
le valeur maximal par la méthode naïve est 23
le valeur maximal par la méthode naïve est 19
```

4.2.1 La solution dynamique avec la programmation dynamique par la méthode Bottom_UP

Pour calculer cette valeur de façon dynamique, nous allons utiliser un tableau bi-dimensionnel de taille $(n+1) \times (W_{max} + 1)$ destiné à contenir les valeurs de $f(k, w)$ pour $k \in [0, n]$ et $W \in [0, W_{max}]$.

Nous prendrons comme valeurs initiales $f(0, W) = f(k, 0) = 0$, et notre but est de calculer $f(n, W_{max})$.

Pour remplir ce tableau, il est primordial de respecter l'ordre de dépendance des cases de ce tableau : la case $f(k, W)$ ne peut être calculée que lorsque les cases $f(k - 1, W)$ et $f(k - 1, W - w_k)$ auront été remplies.



En considérant que les valeurs c_k et w_k sont données sous forme de tableaux, on en déduit l'algorithme :

```
import numpy as np
def sacAdos_Bottom_Up(WL, VL, Wmax):
    n=len(VL)
    f=np.zeros((n+1,Wmax+1))
```

(continues on next page)

(continued from previous page)

```

for k in range(n):
    for w in range(1,Wmax+1):
        if WL[k]<=w:
            f[k+1,w]=max (VL[k]+f[k,w-WL[k]], f[k,w])
        else:
            f[k+1,w]=f[k,w]
return f[n,Wmax]
#return f

```

```

print("le valeur maximal par la méthode bottom up (iteratif) est ",sacAdos_Bottom_
Up(WL,VL,Wmax))

```

```
le valeur maximal par la méthode bottom up (iteratif) est 23.0
```

Remarque. Cet algorithme calcule la valeur maximale qui peut être emportée dans le sac, mais pas la façon d'y parvenir.

Pour la connaître il faut utiliser le tableau (ou le dictionnaire) calculé par la fonction précédente, et retrouver le chemin qui mène de la case initiale à la case finale.

Par exemple, si on modifie la fonction non récursive (la première) pour qu'elle renvoie le tableau f qui a été calculé au lieu de la valeur $f[\text{len}(WL), Wmax]$, le fonction qui détermine les objets à choisir s'écrira :

```

def sacAdos_Bottom_Up_modifi  (WL,VL, Wmax):
n=len(VL)
f=np.zeros((n+1,Wmax+1))
for k in range(n):
    for w in range(1,Wmax+1):
        if WL[k]<=w:
            f[k+1,w]=max (VL[k]+f[k,w-WL[k]], f[k,w])
        else:
            f[k+1,w]=f[k,w]
return f[n,Wmax],f

```

```

def objetsAchoisir(VL, WL, Wmax):
v,f = sacAdos_Bottom_Up_modifi  (VL, WL, Wmax)
sac = []
k, W = len(VL), Wmax
while k > 0:
    if f[k, W] > f[k - 1, W]:
        sac.append((VL[k - 1], WL[k - 1]))
        W -= WL[k - 1]
    k -= 1
return v,sac

```

```
objetsAchoisir(VL, WL, Wmax)
```

```
(15.0, [(10, 6), (2, 9)])
```

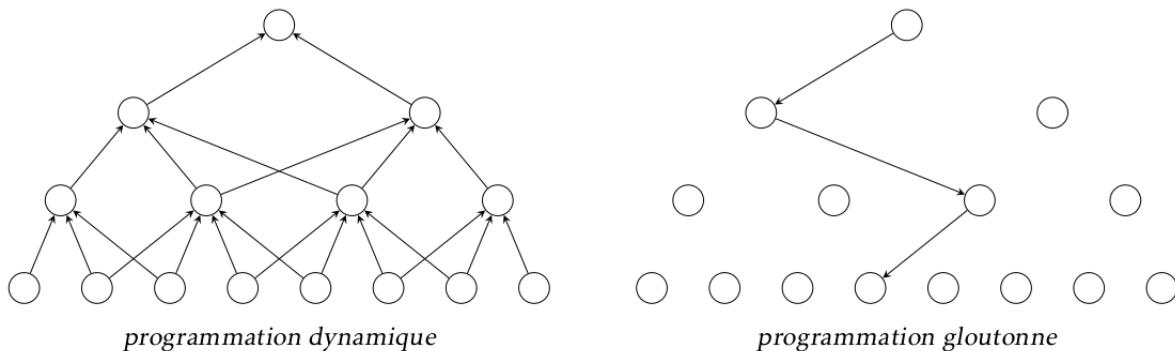
4.2.2 La solution dynamique avec la programmation dynamique par la méthode Top_Down

```
def sacAdos_TD(k,Wmax,memo=dict()):
    if (k,Wmax) not in memo:
        if k===-1:
            memo[(k,Wmax)]=0
            return memo[k,Wmax]
        if Wmax<0:
            memo[(k,Wmax)]=0
            return memo[k,Wmax]
        v,w=VL[k],WL[k]
        if w<=Wmax:
            memo[(k,Wmax)]= max(v+sacAdos_TD(k-1,Wmax-w,memo),sacAdos_TD(k-1,Wmax,memo))
        else:
            memo[(k,Wmax)]=sacAdos_TD(k-1,Wmax,memo)
    return memo[(k,Wmax)]
```

sacAdos_TD(n-1,Wmax)

23

4.2.3 La solution linéaire avec la programmation gloutoune



Pour élaborer un algorithme glouton résolvant le problème, il faut définir une heuristique, ici un critère de priorité pour le choix des objets à prendre.

Nous pouvons par exemple choisir en priorité les objets dont le rapport $\frac{\text{valeur}}{\text{poids}} = \frac{v_i}{w_i}$ est maximal, et remplir le sac tant que c'est possible :

```
def sac_à_dos_glouton(WL,VL,Wmax):
    S=sorted([(WL[k],VL[k]) for k in range(len(WL))],key=lambda e:e[1]/e[0],reverse=True)
    Sol=[]
    w=Wmax
    v=0
    for o in S:
        if o[0]<=w:
            Sol.append(o)
```

(continues on next page)

(continued from previous page)

```
v+=o[1]
w-=o[0]
return v,Sol
#return v
```

```
sac_à_dos_glouton(WL,VL,Wmax)
```

```
(23, [(3, 9), (6, 10), (3, 4)])
```

Cette fois l'algorithme heristique gloutoun donne la solution optimale, mais ce n'est pas le cas en général

Car il donne une solution qui est une proche mais non pas nécessairement égal à la solution optimal.

Pour évaluer la qualité de cette heuristique, On réalise 1000 expériences pour chacune desquelles On :

- pris au hasard 100 objets de valeurs comprises entre 1 et 20 et de poids compris entre 1 et 30 ;
- calcul la valeur optimale obtenue pour un poids maximal égal à 300 à la fois par l'algorithme glouton ci-dessus et par l'algorithme dynamique que nous étudierons plus loin.

```
def fiabilité(glouton,dynamique_bu,samples,number_of_objects):
    VLL=np.random.randint(1,20,(samples,number_of_objects))
    WLL=np.random.randint(1,30,(samples,number_of_objects))
    Wmax=300
    glouton_values=[]
    dynamique_values=[]
    for i in range(samples):
        glouton_values.append(glouton(WLL[i],VLL[i],Wmax)[0])
        dynamique_values.append(dynamique_bu(WLL[i],VLL[i],Wmax))
    return glouton_values,dynamique_values
```

```
g,d=fiabilité(sac_à_dos_glouton,sacAdos_Bottom_Up,200,50)
g=np.array(g)
d=np.array(d)
100*sum(abs(g-d)<0.1)/len(g)
```

```
48.5
```

```
fiab=sum(abs(g-d)<0.1)
```

```
def grand_écart(g,d):
    #l'équivalent au norm sup dans un evn
    imax=0
    max=d[0]-g[0]
    for i in range(len(g)):
        ecart=d[i]-g[i]
        if max<ecart:
            max=ecart
            imax=i
    return imax
i=grand_écart(g,d)
```

```
résultat="""Sur les 200 expériences, {} ont donné le même résultat pour chacun des deux
deux algorithmes,
et dans le cas des {} autres, l'algorithme glouton a toujours rendu un résultat au moins
égal à {}% du résultat de l'algorithme dynamique.
On peut donc considérer ici qu'à défaut de donner un résultat toujours exact, l'algorithme
glouton donne un résultat acceptable tout en ayant une complexité moindre que l'algorithme dynamique.
""".format(fiab,200-fiab,int((g[i]/d[i])*100))
```

```
print(résultat)
```

```
Sur les 200 expériences, 97 ont donné le même résultat pour chacun des deux algorithmes,
et dans le cas des 103 autres, l'algorithme glouton a toujours rendu un résultat au moins
égal à 98% du résultat de l'algorithme dynamique.
On peut donc considérer ici qu'à défaut de donner un résultat toujours exact, l'algorithme
glouton donne un résultat acceptable tout en ayant une complexité moindre que l'algorithme dynamique.
```


LES ALGORITHMES GLOUTONS

5.1 Introduction

La résolution d'un problème algorithmique peut parfois se faire à l'aide de techniques générales (« paradigmes ») qui ont pour avantage d'être applicables à un grand nombre de situations.

Parmi ces méthodes on peut citer par exemple le fameux principe diviser pour régner ou encore la programmation dynamique.

Les algorithmes gloutons, que l'on rencontre principalement pour résoudre des problèmes d'optimisation, constituent l'une de ces techniques générales de résolution.

Nous allons tout d'abord donner une définition intuitive de cette méthode, accompagnée de divers exemples.

5.2 Définition: Algorithme Heuristiques :

- Méthode empiriques simples basées sur l'expérience (résultats déjà obtenus) et sur l'analogie.
- Généralement, on n'obtient pas la solution optimale mais une solution approchée.

5.3 Définition: Algorithme glouton

- Un algorithme glouton est un algorithme qui suit le principe de faire, étape par étape, un choix optimum local, dans l'espoir d'obtenir un résultat optimum global
- Dans les cas où l'algorithme ne fournit pas systématiquement la solution optimale, il est appelé une heuristique gloutonne
- La façon dont on fait le choix est parfois appelée stratégie gloutonne

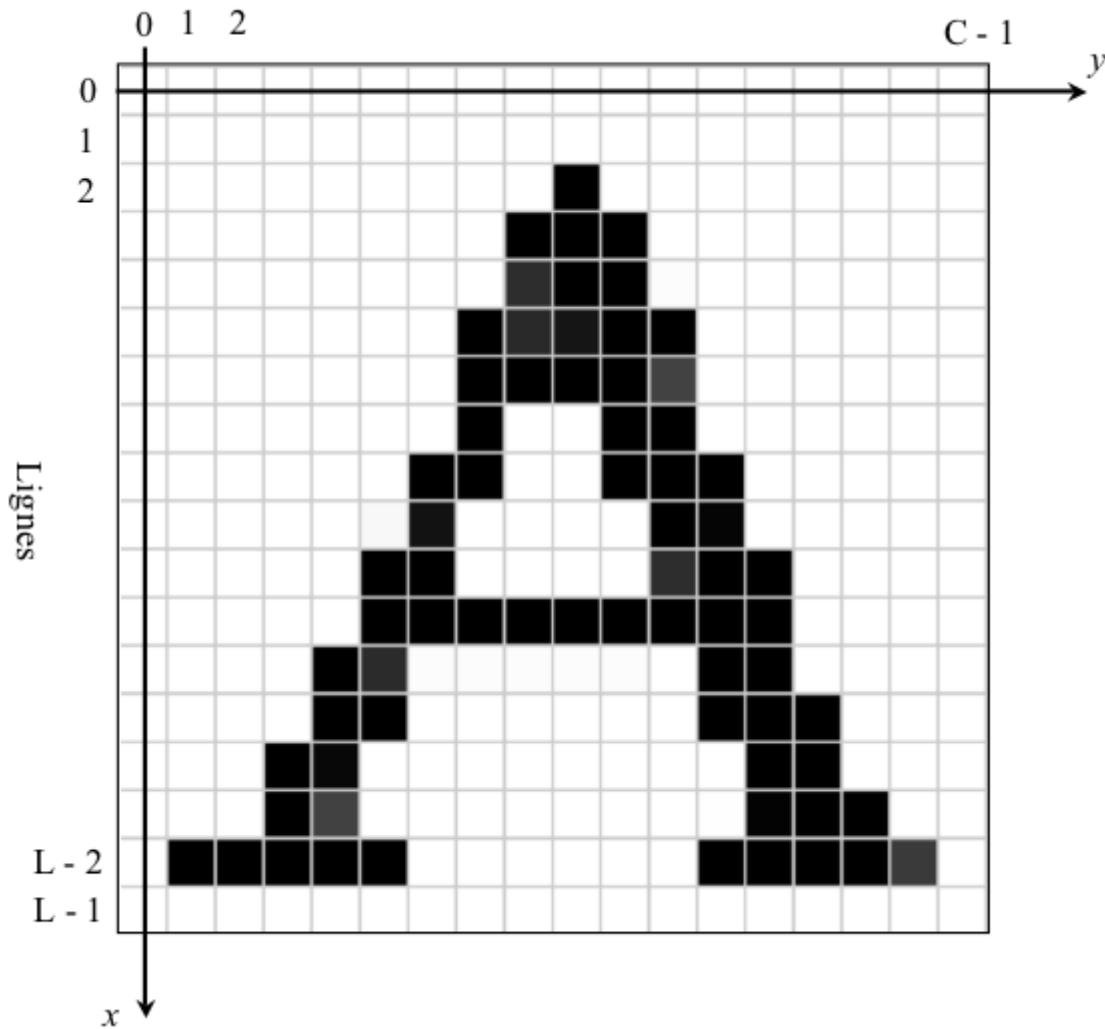
TRAITEMENT DES IMAGES

6.1 Caractéristiques d'une image

6.1.1 Les pixels

Une image est un ensemble de pixels qui peuvent être définis comme des zones carrées identiques de dimensions ($L \times C$). La couleur de chaque pixel est uniforme et l'aspect non granulaire d'une image n'est dû qu'au fait que la taille des pixels est très petite. La figure ci-dessous donne un exemple d'image "binaire" représentant un A. Il n'y a ici que deux couleurs : noir

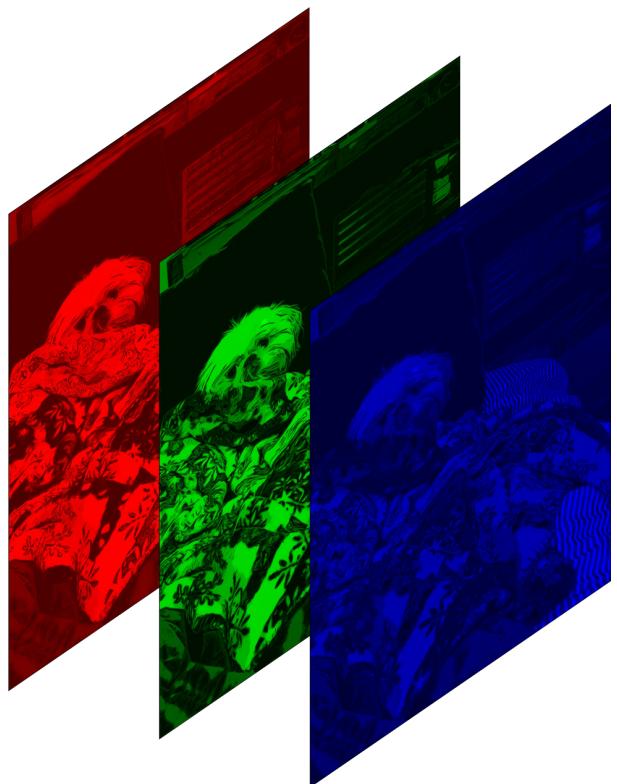
Colonnes



et blanc.

Les images dites en « noir et blanc » sont en fait des images en dégradé de gris. Elles peuvent aussi être représentées par des matrices : chaque élément de la matrice correspondra comme précédemment à un pixel, mais ces éléments seront des nombres entiers donnant l'intensité de gris voulue. On utilise habituellement des nombres de 0 à 255, 0 pour le noir et 255 pour le blanc, soit $256 = 2^8$ niveaux de gris, ce qui permet de coder un pixel par 8 bits.

Les images en couleur sont représentées par trois matrices de taille identique, donnant l'intensité de Rouge, Vert, et Bleu pour chaque pixel. Cette méthode est désignée par le sigle RGB (Red, Green, Blue). Les éléments de ces trois matrices sont encore des entiers de 0 à 255. Dans le système RGB, il y a donc $256^3 = 2^{24} = 16777216$ teintes possibles pour chaque pixel. L'image finale est obtenue en combinant les trois images. Ci-dessous le résultat

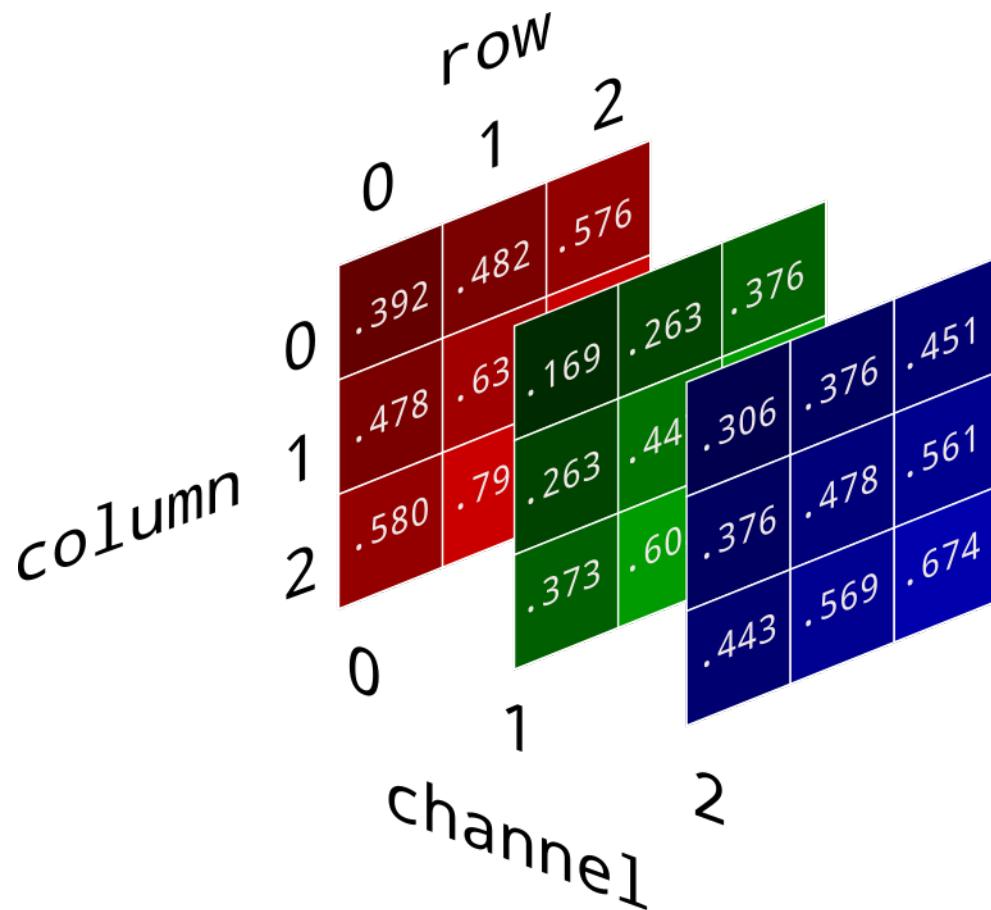


final.

6.1.2 Le format RGB

Une norme fréquemment utilisée pour les images numériques est la norme RGB (Red Green Blue). Ce qu'on appelle un pixel d'écran est composé de trois leds des trois couleurs RGB. Un pixel informatique est alors représenté sous la forme de trois valeurs numériques comprises entre 0 et 255, 0 représentant la led éteinte, et 255 la led totalement allumée. Ainsi :

- la couleur rouge est représentée par (255,0,0) ;
- la couleur verte est représentée par (0,255,0);
- la couleur bleue est représentée par (0,0,255) ;
- la couleur fuchsia (violet) est représentée par (255,0,255) ...



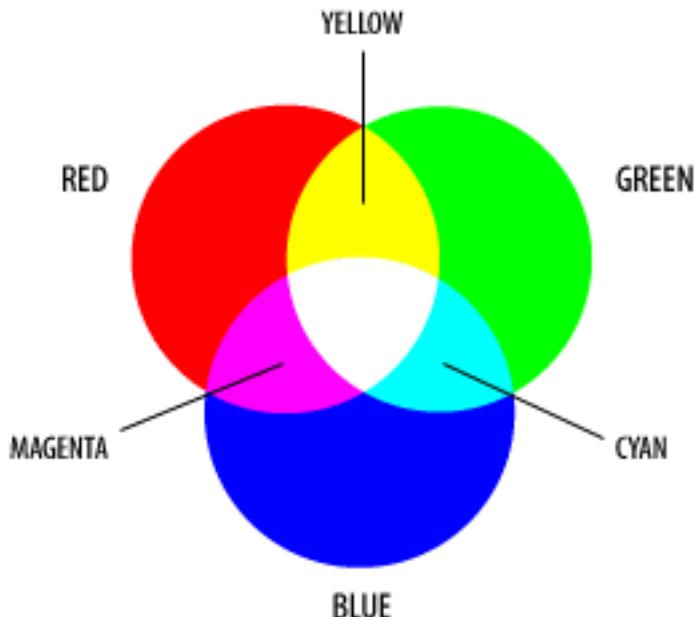
6.1.3 Transparence et Canal Alpha

Pour certains types d'images, un quatrième octet est utilisé pour chaque pixel. Il s'agit du canal alpha, qui caractérise la transparence de ce pixel. Cette couche supplémentaire permet de superposer des images - par exemple dans un jeu vidéo un sprite pourra apparaître sur le fond, sans masquer la totalité sous la forme d'un carré.

Cet octet supplémentaire est utilisé de la manière suivante :

- si il est à 255, le pixel est correctement affiché ;
- si il est à 0, le pixel est transparent.

En informatique, une image en couleurs est constituée d'un ensemble de pixels colorés, rangés en lignes et en colonnes. Chaque pixel est, généralement, constitué d'un mélange de trois couleurs : rouge, vert et



bleu.

6.1.4 Chargement de l'image

Nous allons utiliser le module image de matplotlib pour charger en mémoire une image.

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np
```

On peut procéder de la sorte pour charger une image en-nesyri.png qui se trouverait en répertoire local:

```
img = plt.imread('pictures/en-nesyri.png')
```

La taille de l'image

Si, dans un algorithme, il est besoin de connaître la taille de l'image, on rappelle que `image.shape` fournit un tuple de trois éléments, comprenant la hauteur de l'image, sa largeur, et le nombre de canaux de couleur (3, en principe).

```
print(img.shape)
```

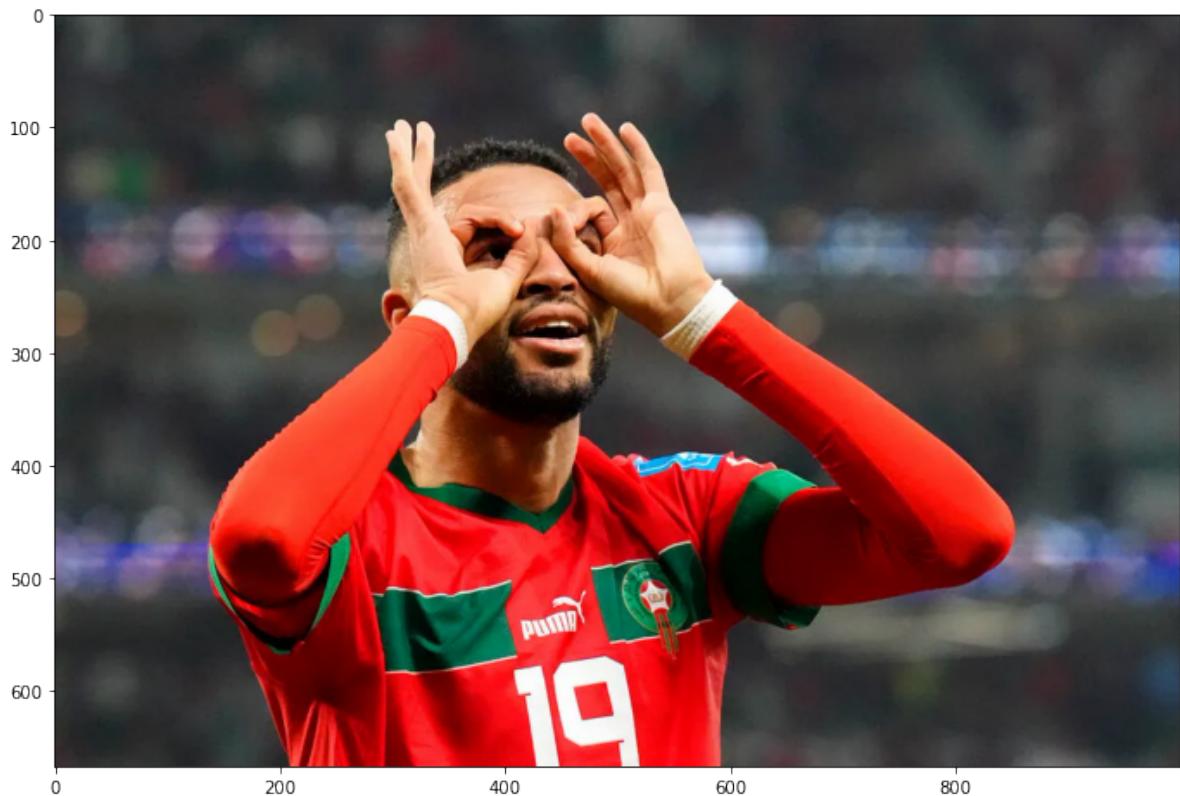
```
(667, 1000, 3)
```

Affichage d'une image

Pour afficher une image ayant trois canaux de couleur, il suffit de faire appel à plt.imshow:

```
plt.figure(figsize=(12, 8))  
plt.imshow(img)
```

```
<matplotlib.image.AxesImage at 0x7f1f01f2a3d0>
```



Ainsi, vous avez vos données dans un tableau numpy (soit en l'important, soit en le générant). Affichons-le, Dans Matplotlib, cela est effectué à l'aide de la fonction imshow(). Ici, nous allons saisir l'objet plot. Cet objet vous permet de manipuler facilement le tracé à partir de l'invite.

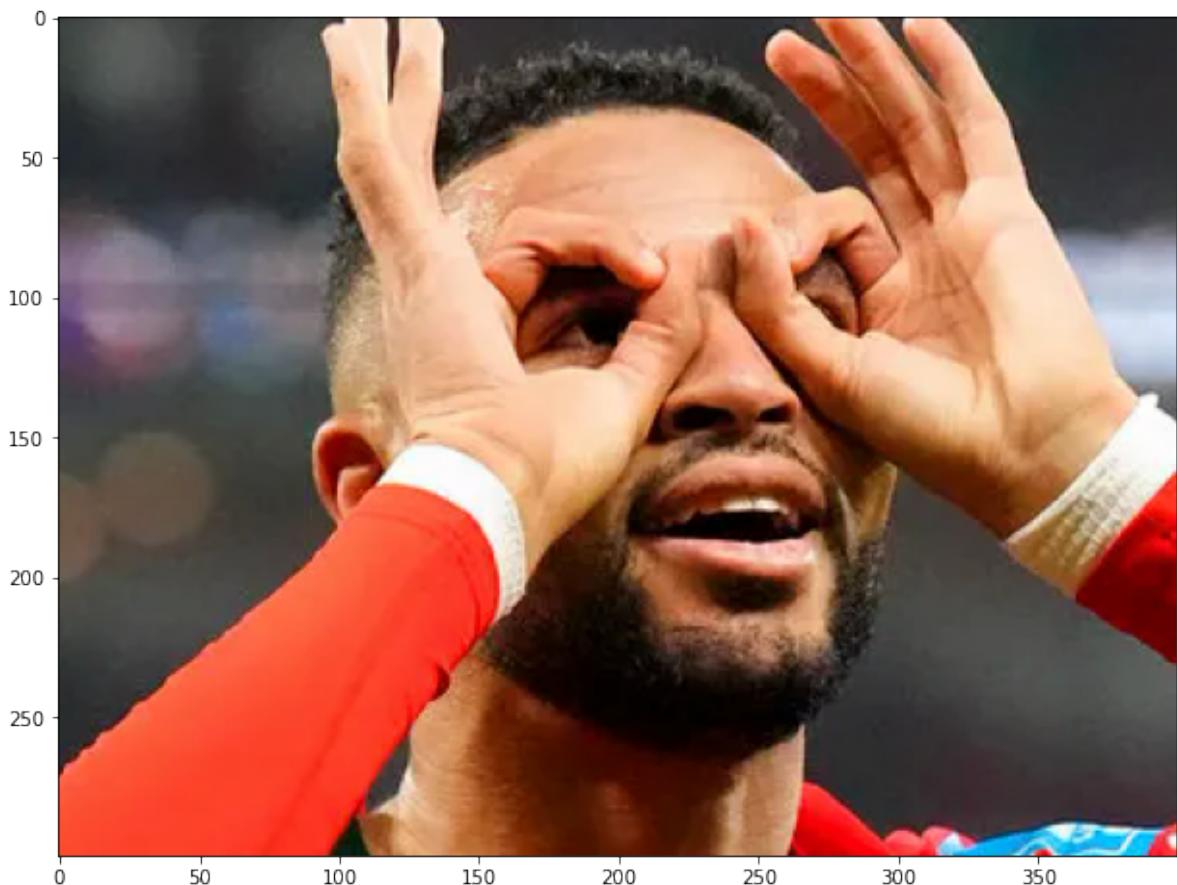
Python utilise un tableau à trois dimensions pour représenter l'image en couleur. Comme pour une matrice, le premier indice désigne la ligne, la seconde la colonne. Le troisième indice désigne le canal de couleur (dans l'ordre, rouge, vert et bleu).

Recadrer l'image

pour sélectionner une partie spécifique en peut utiliser le slicing offert par les objets ndarray de numpy, ainsi ce code permet par exemple de sélectionner la partie visage du joueur Youssef En-nesyri.

```
plt.figure(figsize=(12, 8))  
plt.imshow(img[100:400, 200:600])
```

```
<matplotlib.image.AxesImage at 0x7f1f00b25d60>
```



6.1.5 Modification d'une image

On peut modifier la valeur correspondant à l'intensité d'une couleur d'un pixel par mutation de la case correspondante du tableau, par exemple de la sorte :

```
img[4, 11, 2] = 0.2
```

Cette commande attribue, pour le pixel situé sur la ligne d'index 4 (la cinquième ligne), dans la colonne d'index 11, une intensité égale à 0.2 pour le canal d'index 2 (le troisième canal de couleur, donc, correspondant au bleu). désigne la couleur bleue du pixel situé à la cinquième ligne et à la douzième colonne (on rappelle que les indices débutent à 0). Chacune des valeurs est un entier sur 8 bits, entre 0 et 1. 0 représente l'absence de la couleur concernée, 1 le maximum.\ Si les trois valeurs de couleur sont à 0, on obtient du noir. Si elles sont toutes les trois à 1, du blanc.\ Pour obtenir du rouge, il faut une valeur 1 pour la valeur rouge, et 0 pour les deux autres.

```
img[4, 11] = 0.5, 0.9, 0.2
```

6.2 Traitements basiques de l'image

Une petite fonction qui permet d'afficher deux images old et new

```
def afficher_changement(old,new):
    plt.figure(figsize=(12,8))
    plt.subplot(1,2,1)
    plt.imshow(old)
    plt.subplot(1,2,2)
    plt.imshow(new)
```

Dans un premier temps, nous allons écrire quelques fonctions qui modifient une image, passée en argument de la fonction. Comme les listes, les numpy.array et donc les images sont des objets mutables, que l'on peut directement modifier dans les fonctions. Les fonctions n'ont donc, en principe, pas besoin de retourner de résultat.

6.2.1 Inversion

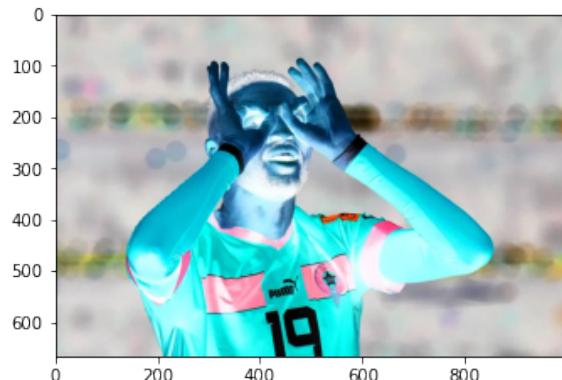
L'inversion d'une image consiste à remplacer, pour chaque pixel et pour chaque canal de couleur, une valeur v par $1.0 - v$ (ou $255 - v$ si l'on travaillait avec des valeurs entières). Cela a pour effet de transformer le blanc en noir (et inversement), le rouge en cyan, le vert en magenta, etc.

Écrire une fonction `Inverse(img)` qui prend en argument une image et l'inverse, et tester son bon fonctionnement.

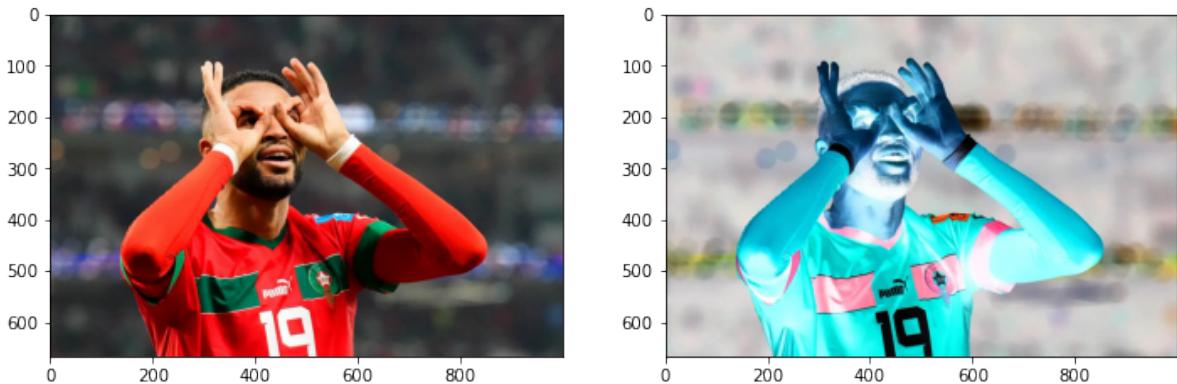
```
def inversion(image):
    return 1-image
```

```
image_negatif=inversion(img)
negatif2=np.ones_like(img)-img
```

```
afficher_changement(img,image_negatif)
```



```
afficher_changement(img,negatif2)
```



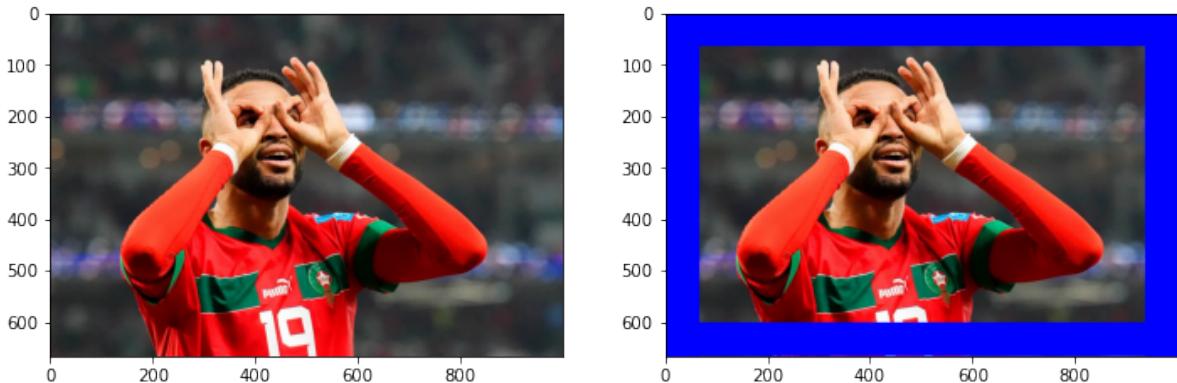
Ajouter un rectangle

- 1.a Écrire une fonction AjouteRectangle(img) qui prend en argument une image, et modifie cette image de façon à ajouter sur celle-ci un rectangle vert de hauteur 80 et de largeur 160 dont le coin en haut à gauche se trouve à la ligne 50 et à la colonne 100.
- 1.b Charger une image, utiliser la fonction précédente, et afficher le résultat à l'écran.

```
from copy import copy
```

```
def border(image):
    newimage=copy(image)
    n=len(image)
    m=int(0.1*n)
    blue=np.array([0,0,1])
    newimage[0:m,:]=blue
    newimage[-m:-1,:]=blue
    newimage[:,0:m]=blue
    newimage[:, -m:-1]=blue
    return newimage
```

```
imageborder=border(img)
afficher_changement(img,imageborder)
```



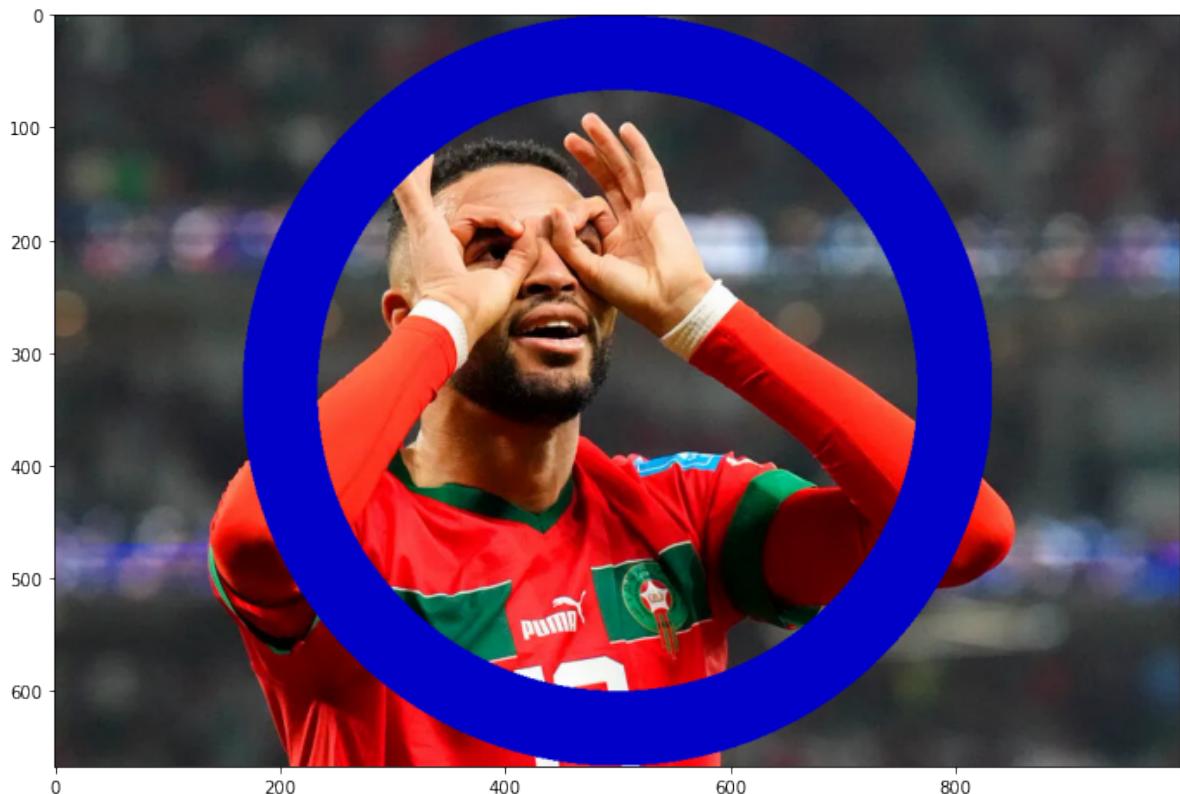
Écrire une fonction AjouteDisque(img) qui prend en argument une image, et modifie cette image de façon à ajouter sur celle-ci un disque noir de rayon égale à la moitié de minimaum entre le nombre des lignes et le nombre des colonnes dont le centre se trouve au centre de l'image, puis tester son bon fonctionnement

```
def norm2(i,j):
    return np.sqrt(i**2+j**2)

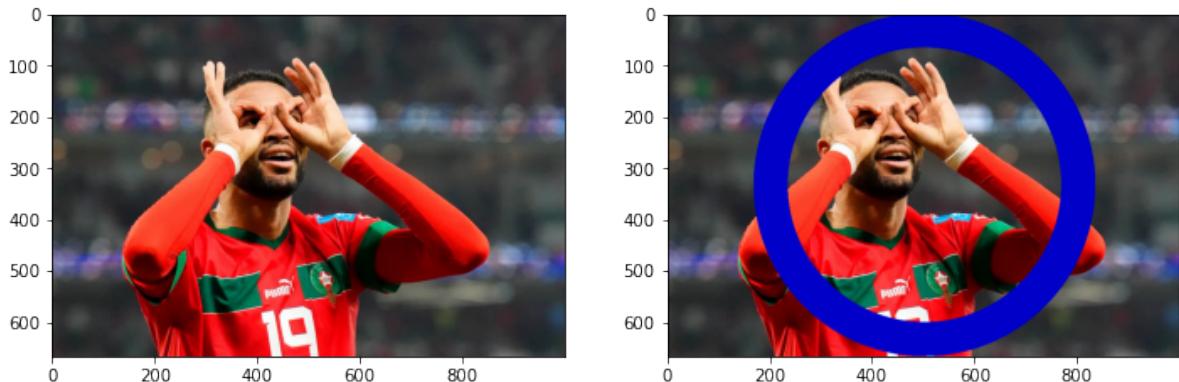
def cercle(image,ep=1):
    mx,my=len(image)//2,len(image[0])//2
    radius=min(mx,my)
    epaisseur=ep*radius
    blue=np.array([0,0,200/255])
    new_image=copy(image)
    masque=[[norm2(i-mx,j-my)<radius and norm2(i-mx,j-my)>radius-epaisseur for j in
    range(len(image[0]))]for i in range(len(image))]
    new_image[masque]=blue
    return new_image
```

```
new_image=cercle(img,0.2)
plt.figure(figsize=(12,8))
plt.imshow(new_image)
```

```
<matplotlib.image.AxesImage at 0x7f1f00093340>
```



```
afficher_changement(img,new_image)
```

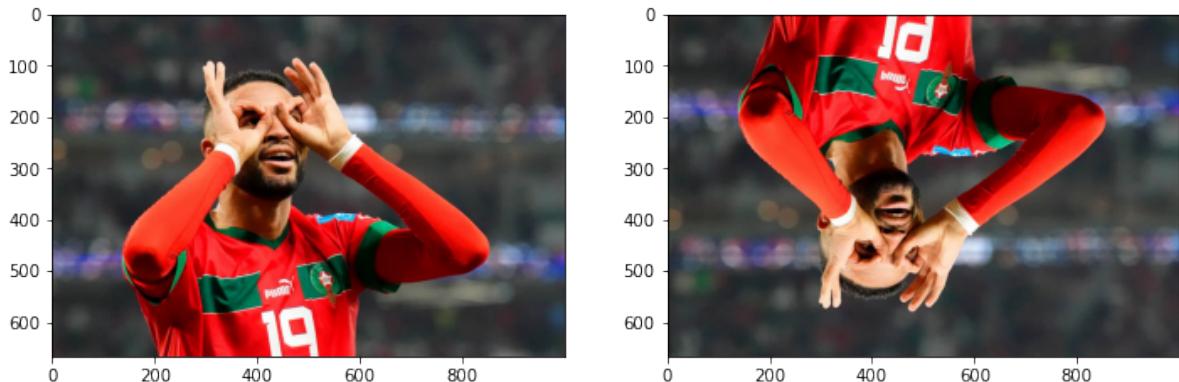


6.2.2 Retournement

Le retournement d'une image consiste à effectuer une symétrie autour de la ligne médiane de l'image. Les pixels de la dernière ligne se retrouvent sur la première ligne (et inversement), et ainsi de suite.

```
def horizontal_sym(image):
    newimage=np.array(list((reversed(image))))
    return newimage
```

```
hsym=horizontal_sym(img)
afficher_changement(img,hsym)
```



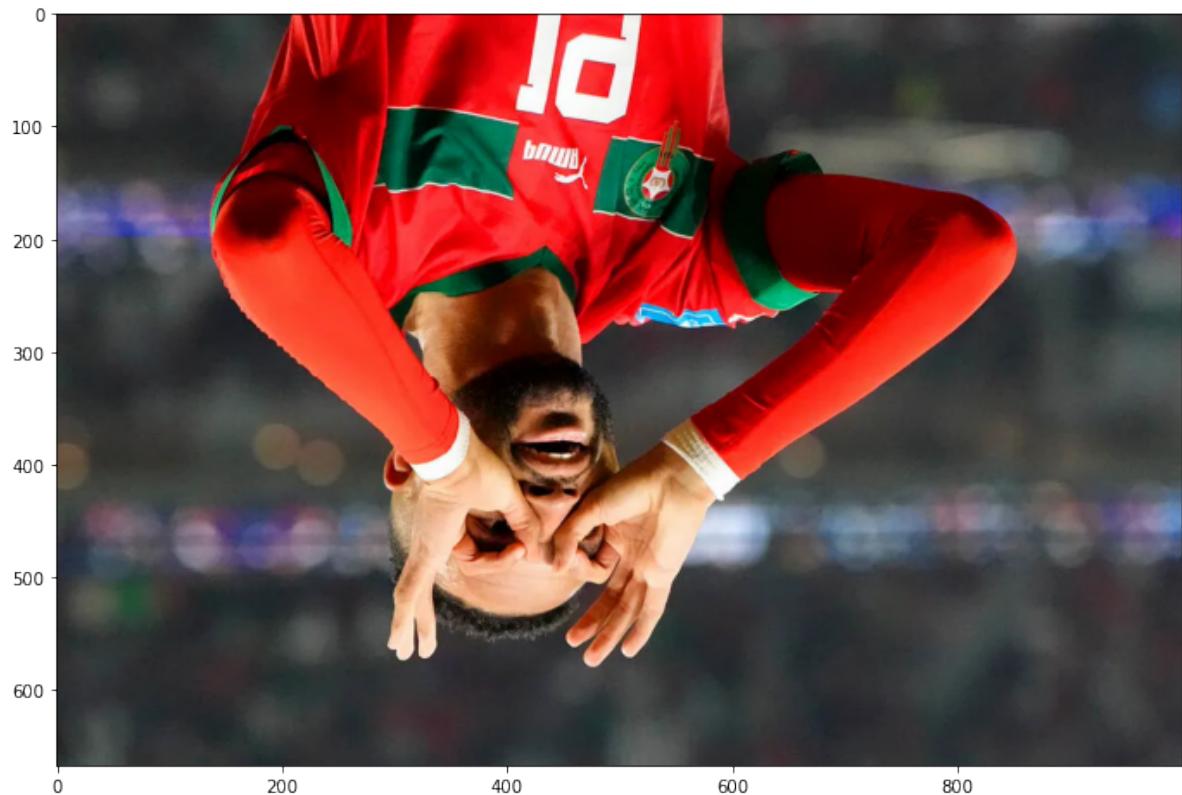
```
def retournemet(image):
    n,_,_=image.shape
    for i in range(n//2):
        l1=image[i]
        image[i],image[n-1-i]=copy(image[n-1-i]),copy(image[i])
        #essayer d'enlever copy
```

```
img=plt.imread("pictures/en-nesyri.png")
```

```
retournemet(img)
```

```
plt.figure(figsize=(12,8))  
plt.imshow(img)
```

```
<matplotlib.image.AxesImage at 0x7f1f01ea4070>
```



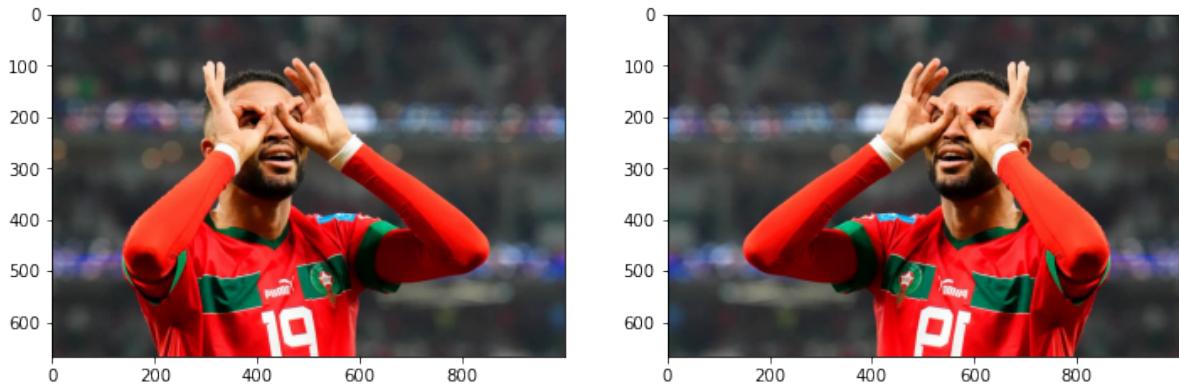
Miroir

Ecrire une fonction Miroir(img) qui renvoie l'image miroir de img.

```
img=plt.imread("pictures/en-nesyri.png")
```

```
def miroir(image):  
    return np.array([[l[i] for i in range(len(l)-1,-1,-1)] for l in image])
```

```
mirror=miroir(img)  
afficher_changement(img,mirror)
```



6.2.3 Passage en niveaux de gris

Pour obtenir une image en niveaux de gris, c'est-à-dire dépourvue de couleur, il suffit que les valeurs des trois canaux de couleur de chaque pixel soient égales. Attention, cela reste une image en couleur, même si tous les pixels sont gris (on n'aura donc pas besoin de l'argument `cmap="gray"` ici).

Pour convertir une image couleur en niveaux de gris, il faut donc remplacer les valeurs des trois canaux par une unique valeur représentant la luminosité du pixel.

Il existe plusieurs façons de calculer cette valeur, selon l'image que l'on souhaite obtenir à l'arrivée. Lorsque l'on souhaite se rapprocher le plus possible de la vision humaine, on peut utiliser la formule suivante (qui privilégie le vert car c'est la couleur à laquelle notre œil est le plus sensible) :

$$0.2126 \times \text{Rouge} + 0.7152 \times \text{Vert} + 0.0722 \times \text{Bleu}$$

On remarquera que la somme des trois coefficients est égale à 1.0, ce qui permet de s'assurer que le résultat ne dépassera pas 1.0, et atteindra la valeur de 1.0 pour un pixel où les trois composantes de couleur égales à 1.0.

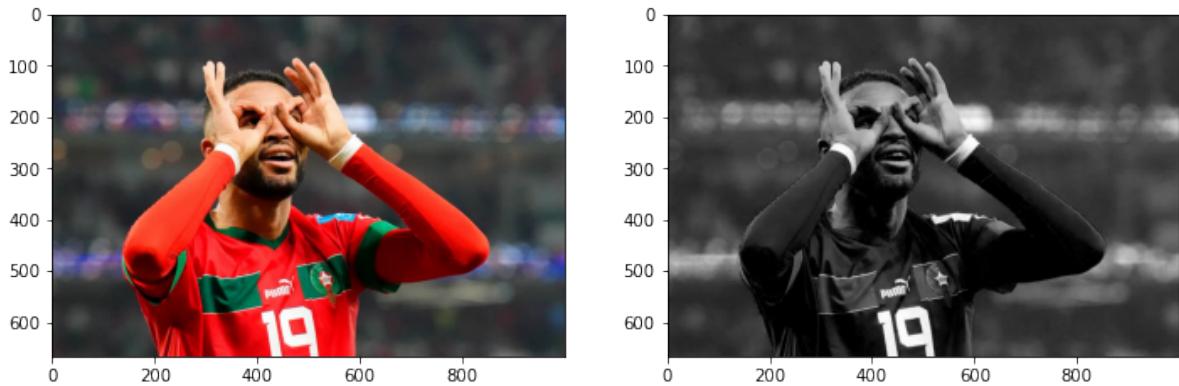
Écrire une fonction `Monochrome(img)` qui prend en argument une image en couleurs et la transforme en une image en niveaux de gris selon la formule précédente, et vérifier son bon fonctionnement.

```
def Monochrome(image):
    newimage=0*image[:, :, 0]+0.7152 *image[:, :, 1]+0.0722 *image[:, :, 2]
    return newimage
```

```
def triplet(image):
    return np.array([[[c]*3 for c in l]for l in image])
```

```
image_grise=triplet(Monochrome(img))
```

```
afficher_changement(img, image_grise)
```



Modification du contraste

Certaines images peuvent être surexposées (trop claire, les valeurs des différents canaux de couleur étant toutes grandes), sous-exposée (même chose avec de petites valeurs), ou peu contrastée (toutes les valeurs sont proches). Corriger le contraste d'une image consiste à modifier la répartition des valeurs entre 0.0 et 1.0 dans chacun des canaux. Cela consiste à appliquer, pour chaque canal de chaque pixel, une fonction définie de la sorte :

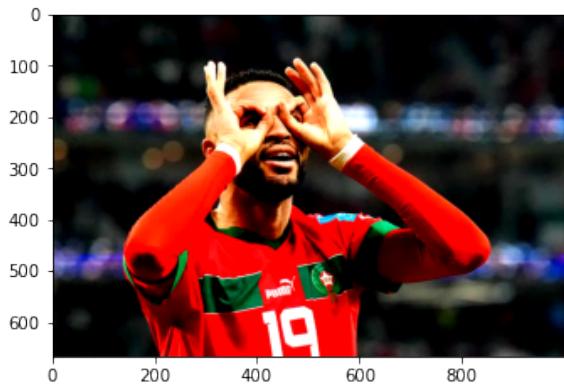
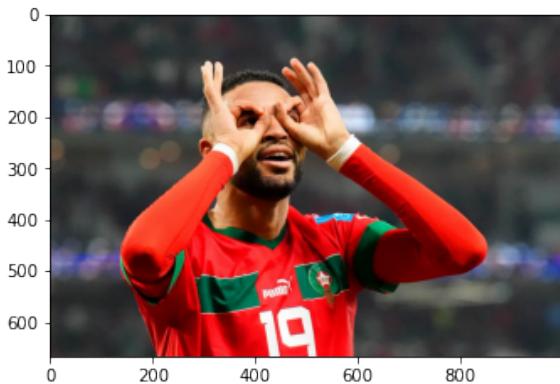
- si la valeur v est inférieure à v_{\min} , on la remplace par 0.0 ;
- si la valeur v est supérieure à v_{\max} , on la remplace par 1.0 ;
- sinon, on remplace la valeur v par $(v - v_{\min}) / (v_{\max} - v_{\min})$.

On supposera, par simplicité, que l'on utilise la même formule pour chacun des trois canaux de couleurs et pour chacun des pixels. Il est fréquent d'utiliser des fonctions différentes pour chaque couleur, ce qui permet de corriger la teinte d'une image (par exemple une image rougie car prise sous un éclairage artificiel). Il arrive parfois que l'on utilise des fonctions différentes selon la zone de l'image, par exemple lorsque l'on veut obtenir un effet « HDR » (haute dynamique). Cela permet de ne pas avoir, sur une même image, des zones sous-exposées et des zones sur-exposées, ce qui est fréquemment le cas pour les images prises à l'extérieur.

Écrire une fonction `AugmenteContraste(img, vmin, vmax)` qui modifie le contraste d'une image, et vérifier son bon fonctionnement. On pourra choisir par exemple comme paramètres $v_{\min} = 0.2$ et $v_{\max} = 0.8$.

```
def augmentContraste(image,vmin = 0.2,vmax = 0.8):
    return np.array([[[0 if v<vmin else 1 if v>vmax else (v-vmin) / (vmax-vmin) for v_
        in c] for c in l] for l in img])
```

```
image_contrast=augmentContraste(img)
afficher_changement(img,image_contrast)
```



```
def red(image):
    newimage=copy(image)
    newimage[:, :, 1:] = 0
    return newimage
```

```
redimage=red(img)
afficher_changement(img, redimage)
```



6.2.4 Histogramme d'une image

L'histogramme d'une image est le graphique qui représente le nombre de pixels existant pour chaque valeur. Calculer l'histogramme de l'image en niveaux de gris, c'est en d'autres termes compter combien il y a de pixels pour chaque nuance de gris.

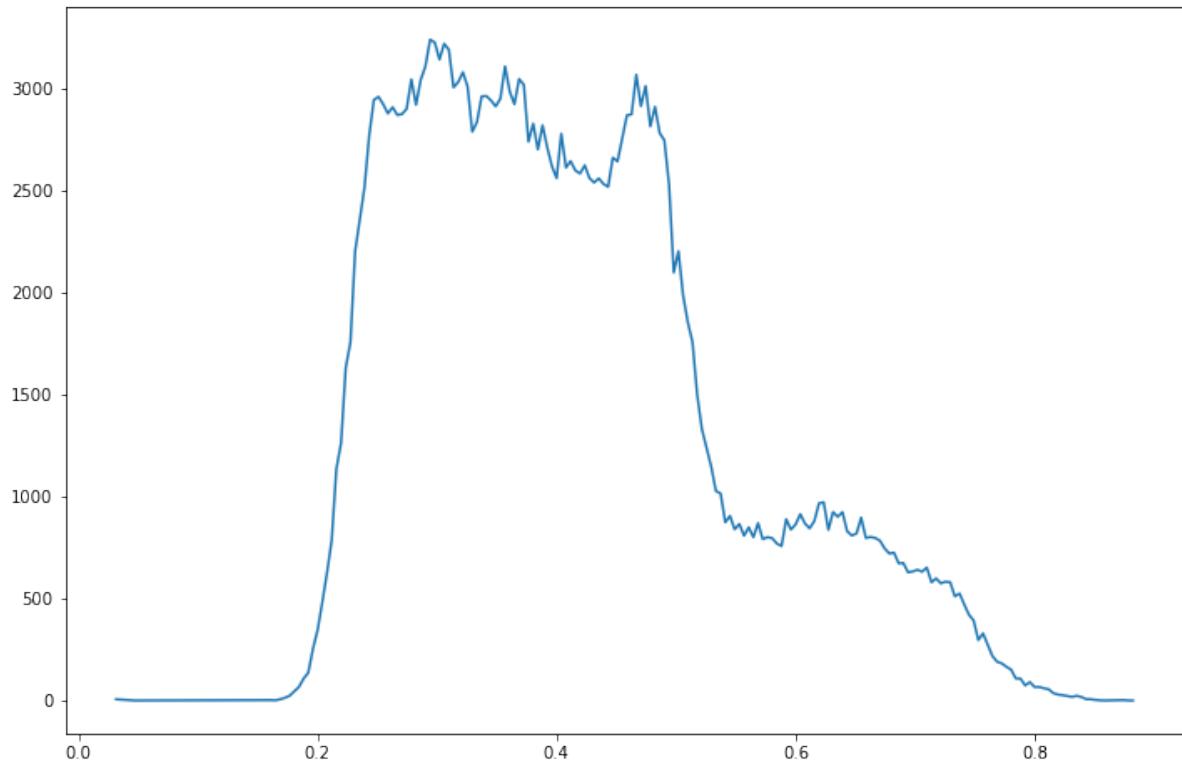
```
def histogram(image_grise):
    histo=dict()
    for l in image_grise:
        for c in l:
            if c not in histo:
                histo[c]=1
            else:
                histo[c]+=1
    return histo.items()
```

```
image2=plt.imread("pictures/Lenna.png")
image_grise2=Monochrome(image2)
histo2=sorted(histogram(image_grise2))
pixels,occurences=np.array(histo2).transpose()
```

```
transp=image_grise.transpose()
```

```
plt.figure(figsize=(12,8))
plt.plot(pixels,occurences)
```

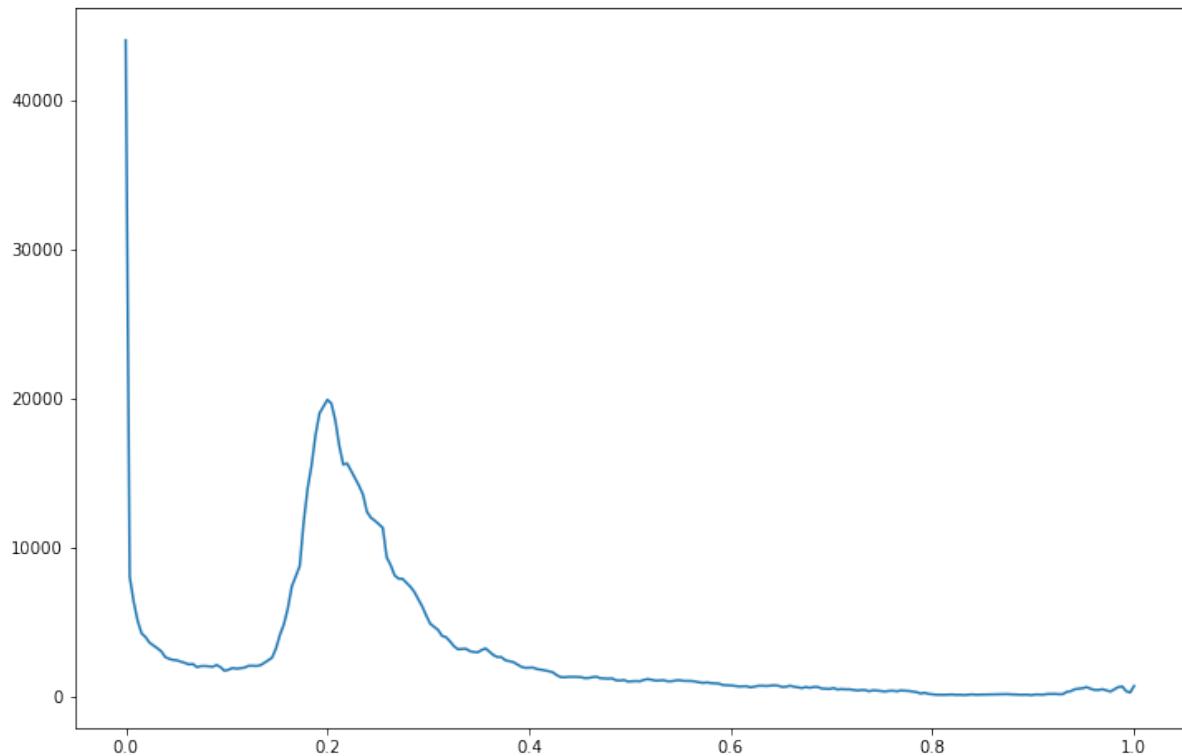
```
[<matplotlib.lines.Line2D at 0x7f1f000d4670>]
```



```
gris=Monochrome(img)
histo1=histogram(gris)
histo1=sorted(histo1)
```

```
i,v=np.array(histo1).transpose()
plt.figure(figsize=(12,8))
plt.plot(i,v)
```

```
[<matplotlib.lines.Line2D at 0x7f1f001acb50>]
```



6.2.5 Seuillage fixe

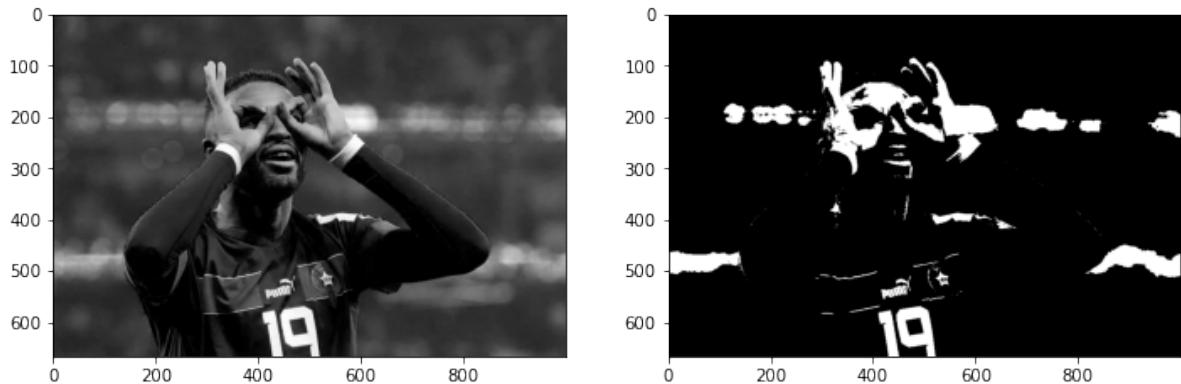
Le seuillage est un traitement qui permet de sélectionner les informations significatives dans une image, ce traitement nécessite le réglage d'un seuil S ,

- Si la valeur d'un pixel $Im[i,j]$ est inférieur au seuil S , alors la valeur de ce pixel est remplacé par 0
- Sinon la valeur de ce pixel est remplacé par 1 Ainsi , on obtient une image en noir et blanc sans niveau de gris

```
def seuillage(image,S=0.5):
    new_image=copy(image[:, :, 0])
    for i in range(len(new_image)):
        for j in range(len(new_image[i])):
            new_image[i, j]=0 if new_image[i, j]<S else 1
    return new_image
```

```
seuillé=triplet(seuillage(image_grise))
```

```
afficher_changement(image_grise, seuillé)
```

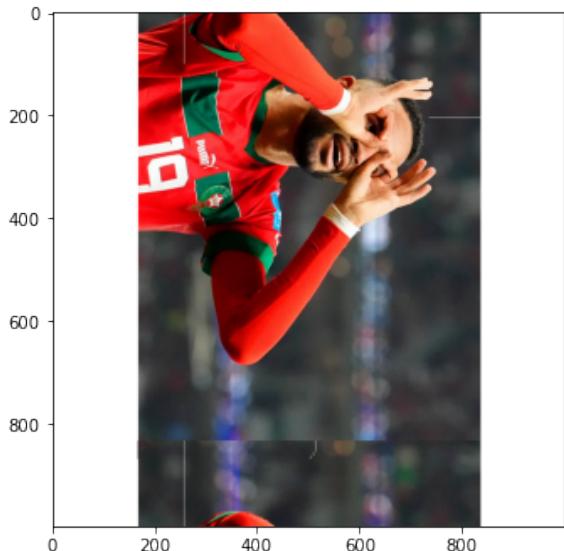


6.2.6 Rotation d'images

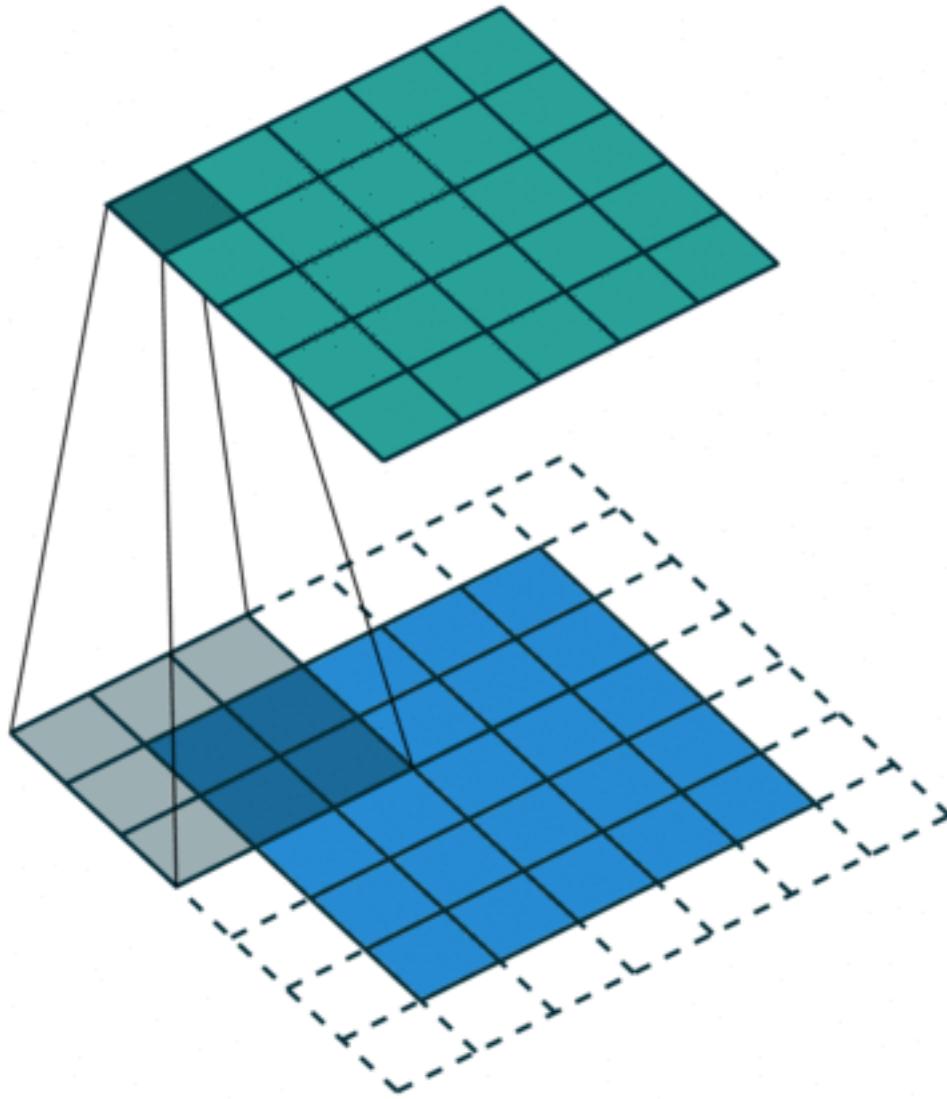
```
def rotate(image, angle=90, centre=None):
    l,c,_=img.shape
    if centre==None:
        centre=np.array([l//2,c//2],)
    new_coordinates=np.ones((l,c,2),dtype="int")
    angle_rad=np.deg2rad(angle)
    rotation_matrix=np.array([[np.cos(angle_rad), -np.sin(angle_rad)], [np.sin(angle_
→rad), np.cos(angle_rad)]])
    for i in range(l):
        for j in range(c):
            new_coordinates[i,j]=centre+rotation_matrix@(np.array([i,j])-centre)
    m=max(l,c)
    new_image=np.ones((m,m,3))
    for i in range(l):
        for j in range(c):
            ii,jj=new_coordinates[i,j]
            new_image[ii,jj]=img[i,j]
    return new_image
```

```
rotate_image=rotate(img,-90)
```

```
afficher_changement(img,rotate_image)
```



6.3 Application de filtres



Un filtre, de manière générale, est une boîte noire qui transforme un signal d'entrée en un signal de sortie. Ce signal d'entrée peut être un signal 1D (tension dépendant du temps), 2D (une image: la valeur du pixel dépend de sa position selon x et y) ou autre. Le signal peut être continu (ses valeurs sont repérées par un paramètre qui varie continûment, comme le temps, c'est le cas de la tension $u(t)$ par exemple) ou discret (ses valeurs peuvent être dénombrées, c'est le cas d'un signal numérique).

Nous avons vu beaucoup de traitements qui s'appliquaient séparément sur chaque pixels. Mais pour obtenir une gamme plus large d'effets, on peut envisager que les nouvelles couleurs d'un pixel dépendent non seulement des anciennes couleurs de ce pixel, mais également des pixels voisins.

Dans cette situation, il n'est pas possible de modifier directement l'image (pourquoi ?) Il faut donc créer une nouvelle image vide, dont on pourra ensuite fixer la couleur de chaque pixel sans toucher à l'image originale.

Les fonctions suivantes vont donc retourner une nouvelle image qui sera le résultat du calcul souhaité, au contraire des

fonctions précédentes qui modifiaient leur argument.

6.3.1 Flou ou Filtre moyenneur

Le filtre moyenneur est une opération de traitement d'images utilisée pour réduire le bruit dans une image et/ou flouter une image.

Le filtre moyenneur fait parti de la catégorie des filtres locaux car pour calculer la nouvelle valeur d'un pixel, il regarde la valeur des pixels prochesConcrètement, la valeur filtrée d'un pixel p est égale à la moyenne des valeurs des pixels proches de p. En général, on définit les « pixels proches de p » comme l'ensemble de pixels contenus dans un carré de largeur k centré sur p :

Pour appliquer un flou à l'image, on remplace chaque couleur de chaque pixel par une moyenne des couleurs des neufs pixels entourant le pixel considéré.

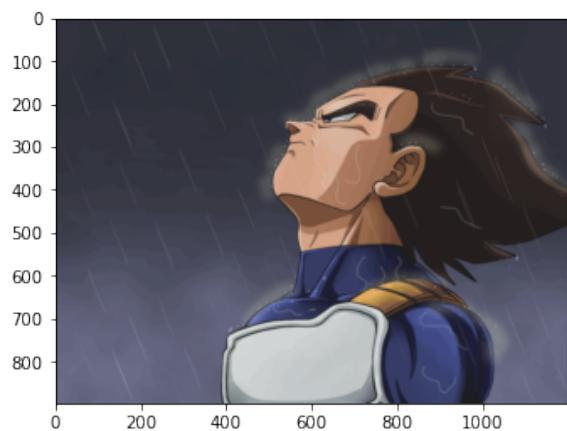
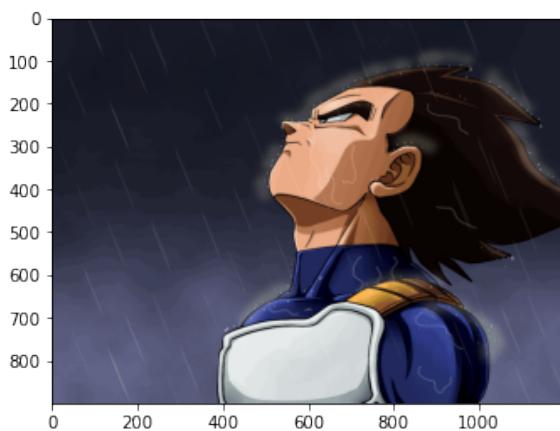
1. Écrire une fonction Floute(img) qui prend en entrée une image img et retourne une nouvelle image correspondant à un flou de l'image originale, et vérifier son bon fonctionnement. On réfléchira à ce qu'il est possible de faire sur les bords de l'image.

```
vegeta_image=plt.imread("pictures/vegeta.png")
```

```
def flou(image):
    flou=(image[2:,:2:]+image[2:,1:-1]+image[2:,:-2]+\n        image[1:-1,2:]+image[1:-1,:-2]+\n        image[:-2,2:]+image[:-2,1:-1]+image[:-2,:-2])/9
    return flou
```

```
flou_image=flou(vegeta_image)
```

```
afficher_changement(vegeta_image, flou_image)
```



```
def flou_loops(image):
    newimage=np.zeros_like(image)
    l,c,_=image.shape
    for i in range(1,l-1):
        for j in range(1,c-1):
```

(continues on next page)

(continued from previous page)

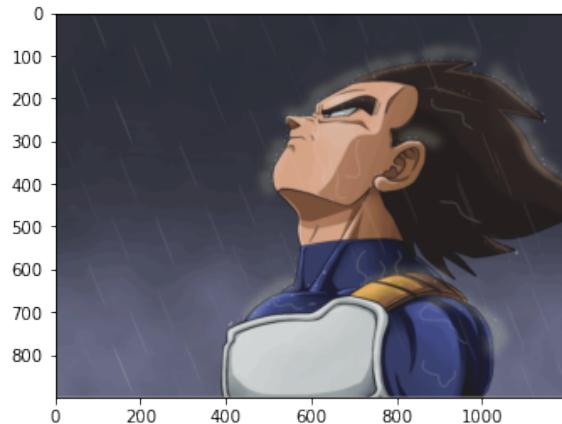
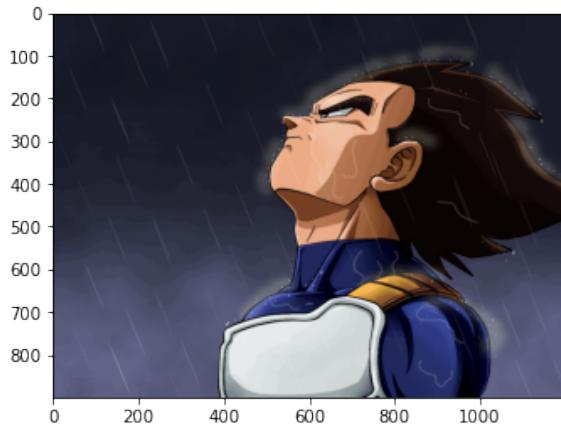
```

newimage[i,j]=(image[i-1,j-1]+image[i-1,j]+image[i-1,j+1]+image[i,j-
→1]+image[i,j+1]+image[i+1,j-1]+image[i+1,j]+image[i+1,j+1])/9
return newimage

```

```
image2_flou=floou_loops(vegeta_image)
```

```
afficher_changement(vegeta_image,image2_flou)
```



6.3.2 Détection de contours

Un contour, dans une image, est une zone de fortes variations des valeurs: si on pense à la photographie d'un poster sombre sur un mur blanc, le contour sera l'endroit où l'on passe du sombre au blanc. Détecter des contours est une opération importante, c'est la première étape vers des opérations de comptage automatisé, de reconnaissance de visages, etc. C'est une opération relativement compliquée dont nous allons voir ci-après quelques bases.

Il est possible de détecter les contours d'une image en déterminant, pour chaque pixel, la norme du gradient en ce point. En effet, un gradient élevé correspond à une brusque variation de la luminosité ou de la couleur, ce que l'on trouve en général au bord des objets.

```

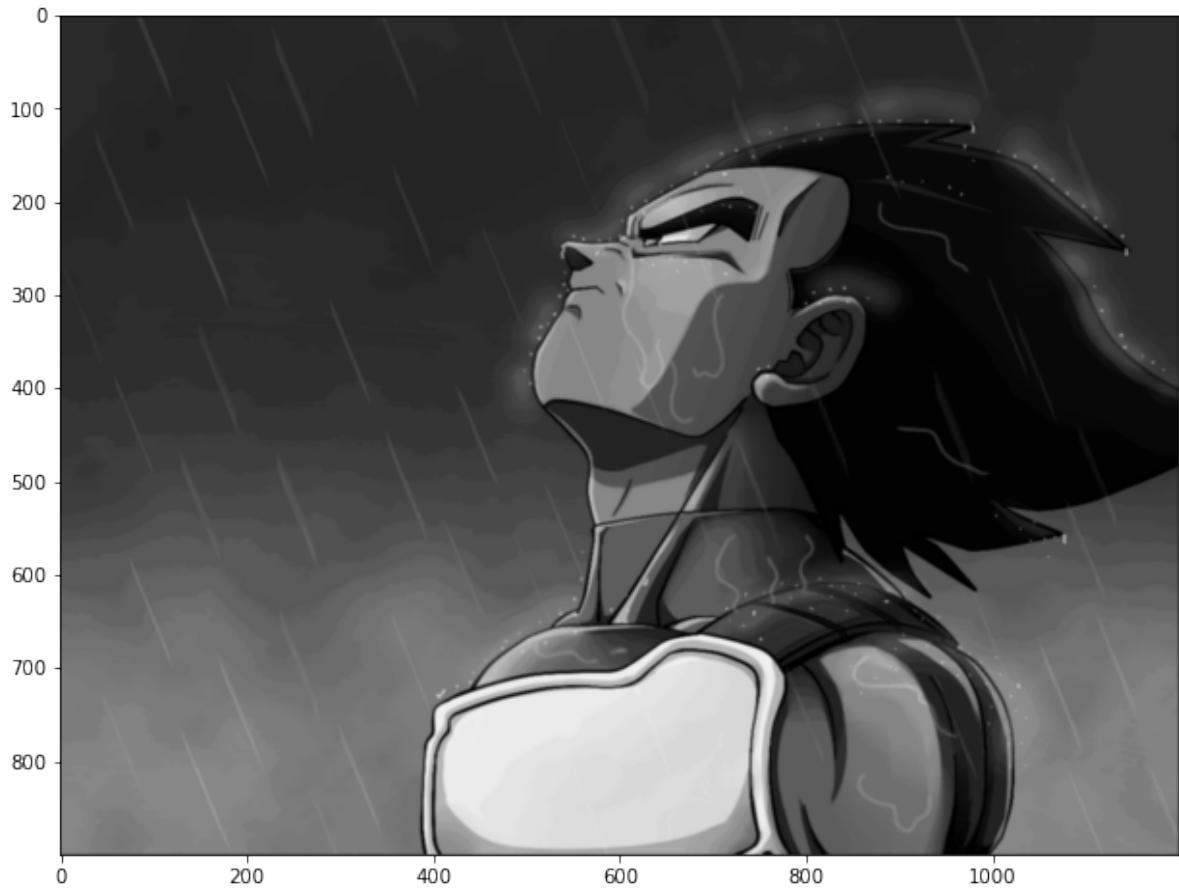
def convolution(image,noyeau):
    nl,nc=noyeau.shape
    l,c=image.shape
    image_augmentée=np.zeros((l+2,c+2))
    image_augmentée[1:-1,1:-1]=image
    newimage=np.zeros_like(image)
    for i in range(l):
        for j in range(c):
            newimage[i,j]=abs(sum(sum(noyeau*image_augmentée[i:i+3,j:j+3])))
    return newimage

```

Concolution et filtre Moyenneur

```
moyenne=np.array([[1,1,1],[1,1,1],[1,1,1]])/9
gris=Monochrome(vegeta_image)
f1=convolution(gris,moyenne)
f1=triplet(f1)
plt.figure(figsize=(12,8))
plt.imshow(f1)
```

<matplotlib.image.AxesImage at 0x7f1f0029b8e0>

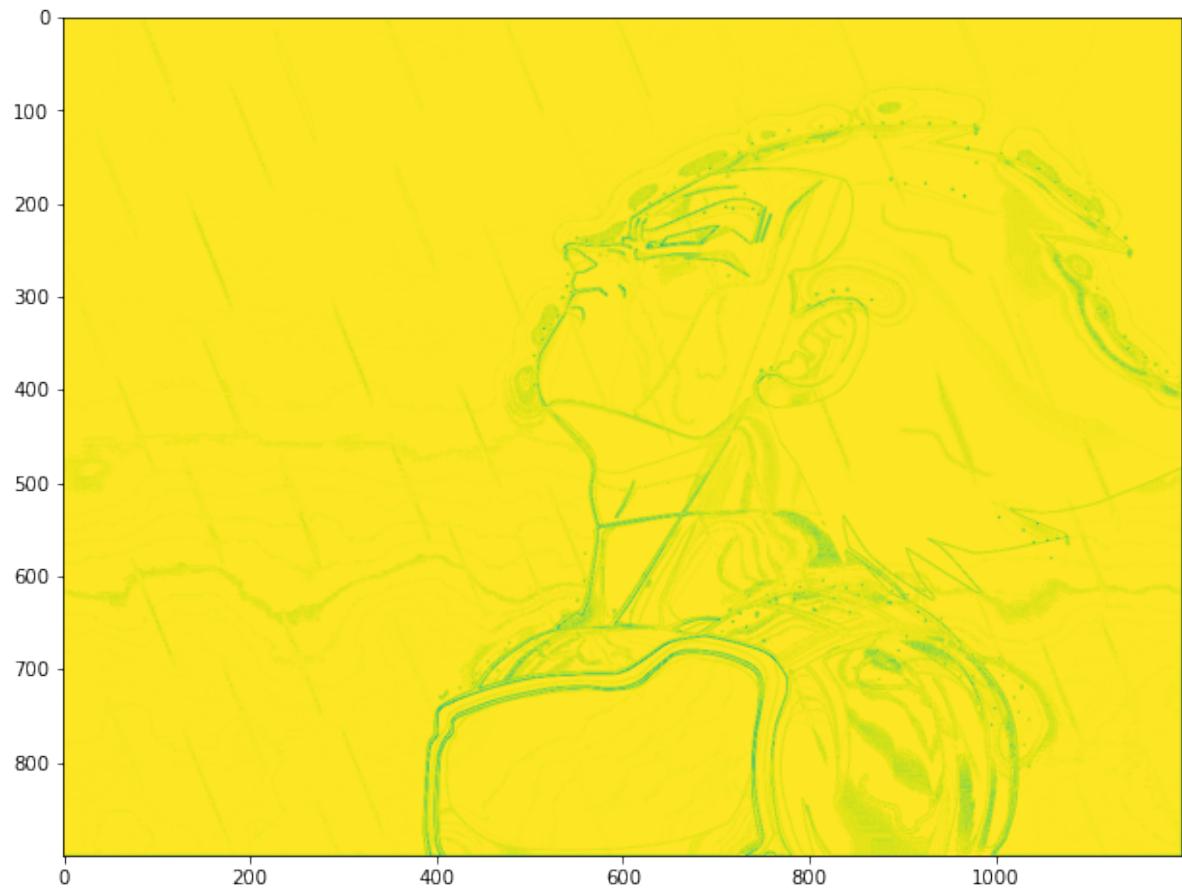


6.3.3 Détection des contours

```
gris=triplet(Monochrome(vegeta_image))
gris=gris[:, :, 0]
```

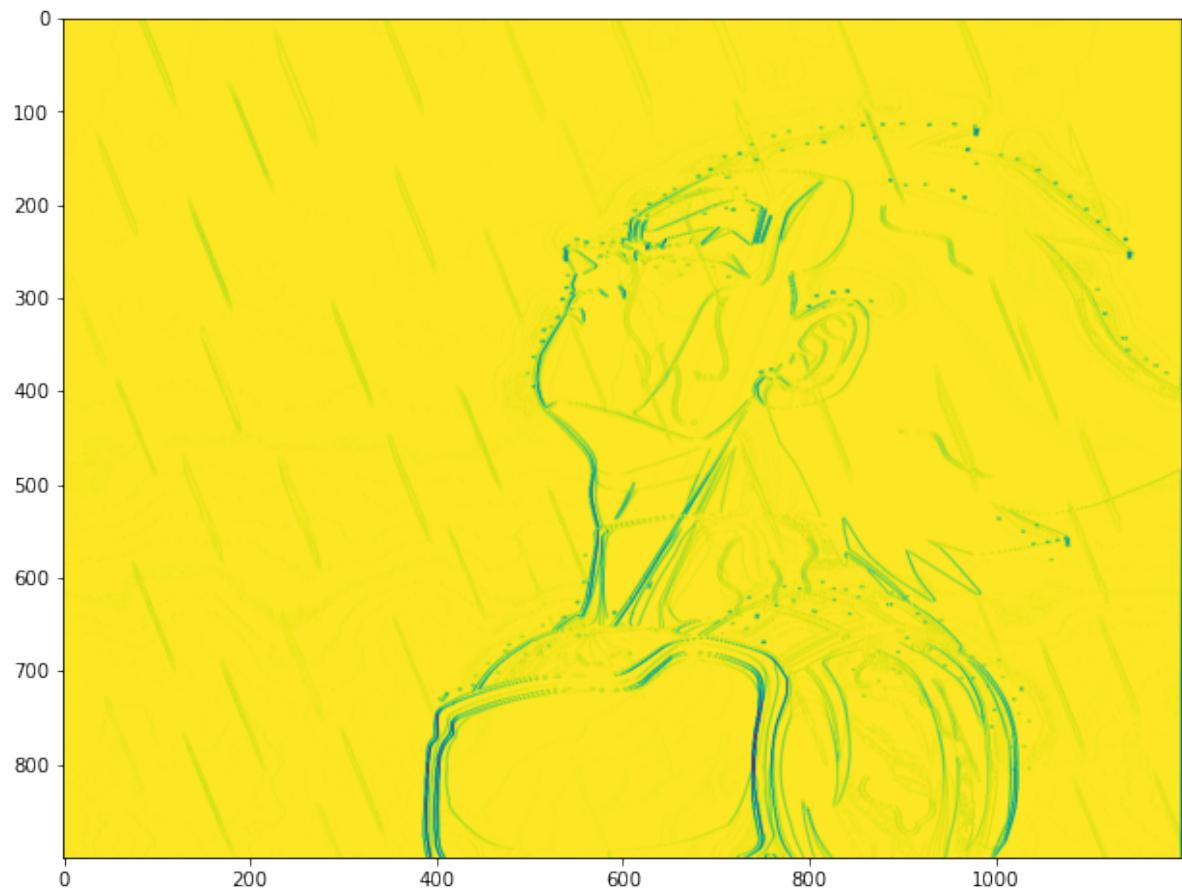
```
diff=np.array([[0,1,0],[1,-4,1],[0,1,0]])
f1=convolution(gris,diff)
plt.figure(figsize=(12,8))
plt.imshow(1-f1)
```

```
<matplotlib.image.AxesImage at 0x7f1f000c95b0>
```



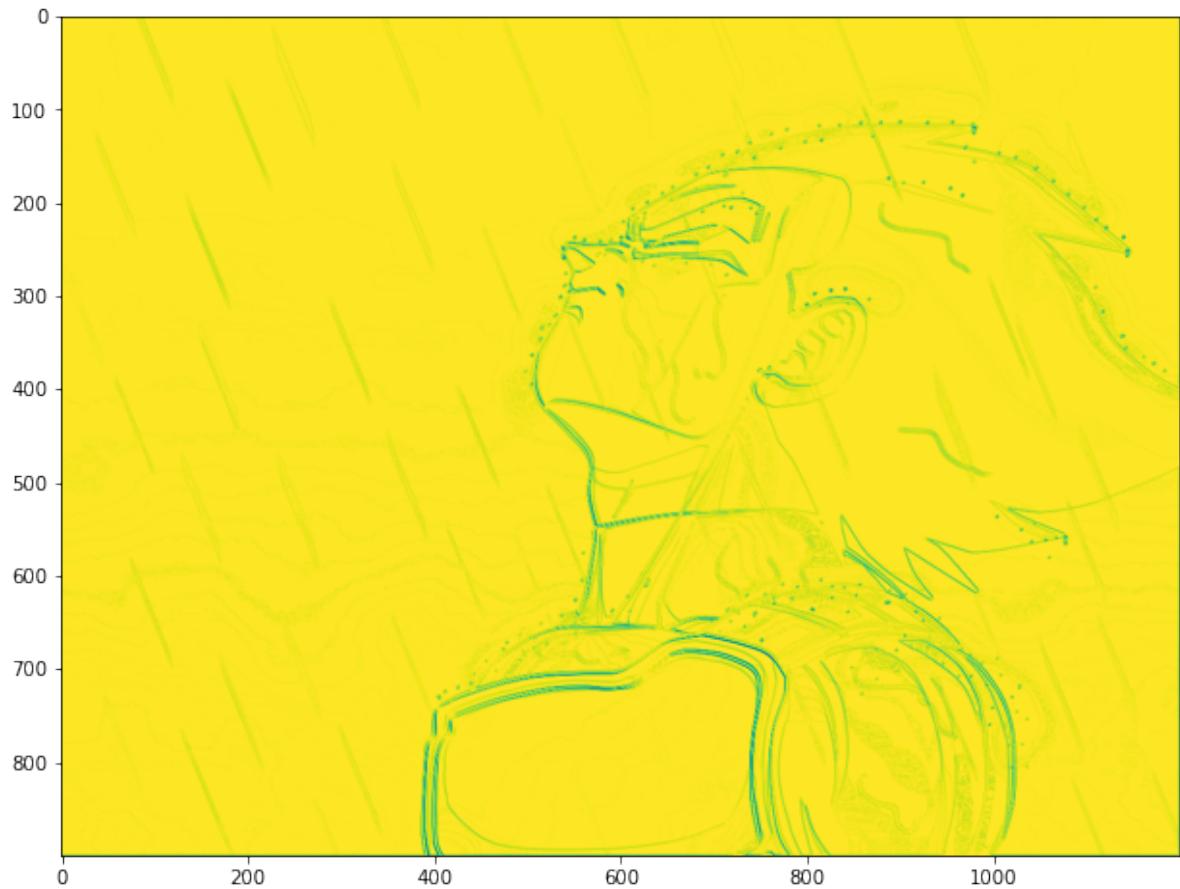
```
sobel=np.array([[-1,0,1],[-2,0,2],[-1,0,1]])
sbl=convolution(gris,sobel)
plt.figure(figsize=(12,8))
plt.imshow(1-sbl)
```

```
<matplotlib.image.AxesImage at 0x7f1efaf4bfd0>
```



```
sobel2=np.array([[0,1,0],[-1,0,1],[0,-1,0]])
sbl2=convolution(gris,sobel2)
plt.figure(figsize=(12,8))
plt.imshow(1-sbl2)
```

```
<matplotlib.image.AxesImage at 0x7f1f0012e310>
```



```
import numpy as np
import matplotlib.pyplot as plt
from collections import Counter
```

```
#https://www.youtube.com/watch?v=0p0o5cmgLdE
```

Dans ce chapitre, on voit deux algorithmes pour l'intelligence artificielle : les k plus proches voisins (KNN k nearest neighbors) et les k-moyennes (k means).

ALGORITHME KNN: COURS

7.1 Classification supervisée

- On considère le problème suivant : on dispose N objets sur lesquels on a mesuré différentes valeurs. Les mêmes mesures ont été faites sur chacun des objets.
- La valeur de N est très grande.
- Ces objets se répartissent selon T classes, avec T petit. On sait dans quel classe va chaque objet que l'on a mesuré.
- Maintenant, on ajoute un nouvel objet sur lequel on a fait les mêmes mesures.
- On souhaite lui associer une classe.
- De quelle classe est-il le plus proche ?

On parle ainsi **D'APPRENTISSAGE SUPERVISÉ**. En effet, on apprend sur un jeu des données déjà classées comment classer les nouvelles données.

Exemple 1.

- Ce type d'algorithme est très utilisé en intelligence artificielle particulièrement dans le commerce en ligne.
- On a des informations sur N personnes (âge, sexe, habitude de navigation, etc.). On a classé leurs centres d'intérêts selon T types.
- On peut alors proposer automatiquement à un nouvel individu des publicités adaptées en le classant automatiquement.

Exemple 2.

- Pour faire des diagnostics automatiques, On dispose de mesures médicales sur N individus pour lesquels on sait si il y a ou pas une tumeur cancéreuse, c'est-à-d que l'on a répartis dans deux classes : malade / sain.
- À partir de ces données, on peut déterminer automatiquement si un individu doit être considéré comme malade ou sain.

7.2 La notion de distance:

- Pour donner un sens à la classe la plus proche, cela suppose à minima de choisir une manière de mesurer les écarts entre les objets, c-à-d d'avoir une distance sur les mesures associées à aux objets.
- Cela n'est pas du tout évident. D'autant plus que les mesures peuvent prendre diverses formes.
- Le choix de la distance est donc critique pour la classification.

```
d1=lambda A,B: sum([abs(a-b) for a,b in zip(A,B)])
d2=lambda A,B: np.sqrt(sum([(a-b)**2 for a,b in zip(A,B)]))
d_inf=lambda A,B: max([abs(a-b) for a,b in zip(A,B)])
d2_np=lambda A,B: np.sqrt(sum((A-B)**2))
d1_np=lambda A,B: sum(abs(A-B))
d_inf_np=lambda A,B: max(abs(A-B))
```

7.3 Principe de l'algorithme

Mathématiquement,

- On dispose donc d'un ensemble E assez complexe (l'ensemble des mesures), E est muni d'une distance d .
- On dispose d'un ensemble C de classe de cardinal T .
- On dispose donc de N éléments de $E \times C$ (les mesures faites sur les objets et les classes associées). On les notes $(x_j, c_j)_{j \in [1, N]}$.
- Ainsi, x_j est la liste des mesures faites sur l'objet j , et c_j la classe associée.
- **Problème** On considère de plus un élément y de E , non classé et on veut lui associer une classe.

Exemple : Comme problème jouet, on considère donc que $E = \mathbb{R}^n$ et que l'on prend la distance euclidienne.

L'algorithme des KNN consiste à :

- mesurer les distance $d(y, x_j)$ pour $j \in [|1, N|]$,
- déterminer les k plus proches voisins de y . Les k éléments les plus proches de y sont notés : $x_{i_1}, x_{i_2}, \dots, x_{i_k}$.
- Choisir pour classe de y la classe la plus fréquente parmi celles de $(x_{i_1}, x_{i_2}, \dots, x_{i_k})$

Le choix du paramètre k (nombre de voisins que l'on prend en compte) est critique. Il n'est pas facile de régler ce paramètre et de l'interpréter. D'autre part, on peut avoir plusieurs classes avec le même effectif parmi les k voisins. Dans ce cas, l'algorithme ne permet pas de conclure et on attribue généralement arbitrairement une classe au nouvel objet y .

7.4 Mise en place de chaque étape

7.4.1 Mesure des distances

On commence donc par créer une liste des distances : $d(y, x_j)$ pour $j \in [|1, N|]$.

```
def Distances(y ,LearningData,d=d2) :
    """
    entrée :
        lX = liste d'objets (élement de R^n) de longueur N
```

(continues on next page)

(continued from previous page)

```

y = objet (élement de R^n)
on dispose d'une fonction d
qui mesure la distance entre objets
sortie :
    lD = liste de float= liste des distances d(y,xk) pour k dans [|1 , N |]
"""
return [d(y,x) for x in LearningData]

```

7.4.2 Déterminer les k plus proches voisins

Ensuite, il s'agit de déterminer les k plus proches voisins. Pour cela, on se contente d'utiliser l'algorithme de tri rapide en utilisant la fonction **sorted**.

Ici, on voit que l'on veut les indices des éléments les plus proches, pas les valeurs. En effet, ce n'est pas trier les distances qui nous intéressent mais trier les points. C'est pour cette raison nous n'allons pas utiliser **sorted** directement sur les distances mais nous allons utiliser la fonction **np.argsort** qui retourne la liste des indices des éléments qui rendent la liste des distances une liste triée.

```

## Exemple
distances=[5,10,9,11,0.3]
indices=np.argsort(distances)
print("les distances: ",distances)
print("les indices: ",indices)
print("{} est l'indice de {} dans la liste distances, ce qui représente la distance"
      "minimale,\nainsi {} est l'indice de la distance maximale qui est {}".
      format(indices[0],distances[indices[0]],indices[-1],distances[indices[-1]]))

```

```

les distances: [5, 10, 9, 11, 0.3]
les indices: [4 0 2 1 3]
4 est l'indice de 0.3 dans la liste distances, ce qui représente la distance
  minimale,
ainsi 3 est l'indice de la distance maximale qui est 11

```

```

def KNN(Ditanceslist,k):
    """
    entrée :
        Ditanceslist = liste de float
            = liste des distances entre l'objet à placer
            y et chacun des objets de la liste d'apprentissages
        k = int = nombre de voisins à prendre en compte
    sortie : lKNN = liste de int de longueur K
            = liste des indices des K plus proches voisins .
    """
    return np.argsort(Ditanceslist)[:k]

```

```

def KNN_version2(Ditanceslist,k):
    triee=sorted(enumerate(Ditanceslist),key=lambda e:e[1])
    return [e[0] for e in triee][:k]

```

7.4.3 Choisir la classe la plus fréquente

Pour déterminer la classe la plus fréquente, il faut compter combien de fois est présente chaque classe.

Pour cela, on parcourt les k plus proches avec un compteur de longueur T (le nombre de classe).

Pour chaque élément lu, on incrémente le compteur correspondant.

```
def Compter(TargetValues):
    compteur={}
    for e in TargetValues:
        if e not in compteur:
            compteur[e]=1
        else:
            compteur[e]+=1
    return compteur
```

Ensuite, un simple algorithme de recherche de maximum donne la classe la plus fréquente.

```
def Plus_commun(compteur):
    max(compteur, key=lambda e:compteur[e])
```

7.4.4 Prédire la Sortie d'une nouvelle donnée

```
def predict(y, LearningData, LearningTarget, d=d2_np, k=11):
    Distanceslist=distances(y, LearningData, d)
    voisins=KNN(Distanceslist, k)
    return Plus_commun(Compter(LearningTarget[voisins]))
#return Counter(LearningTarget[voisins]).most_common()[0]
#
```

7.4.5 Matrice de confusion

La matrice de confusion permet de mesurer la qualité du système de classification. Pour tester la qualité, on prend M objets dont on connaît la classification. Cette classification est qualifiée de certaine. On applique l'algorithme à chacun de ses M objets et on note la classification obtenue (dite classification estimée). On met ces résultats dans une matrice : chaque ligne de la matrice correspond à une classe certaine, chaque colonne à une classe estimée. Ainsi ligne i et colonne j on met le nombre d'éléments qui ont été classifiés dans la classe j alors qu'ils sont dans la classe i. Si la classification était parfaite, alors seule la diagonale aurait des éléments non nuls. On considère qu'une classification est de qualité lorsque chaque ligne contient 95% de ses valeurs sur l'élément diagonal.

```
def ConfusionMatrix(TestData, TestTarget, model):
    LearningData, LearningTarget, d, k=model
    types=list(set(TestTarget))
    n=len(types)
    matrice=np.zeros((n,n))
    for i in range(len(TestData)):
        p=predict(TestData[i], LearningData, LearningTarget, d, k)
        matrice[TestTarget[i],p]+=1
```

TP IRIS

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from MyKNNLIB import *
from MyKNNLIB import predict
```

On importe la célèbre base de données iris. Elle contient des informations sur 3 variétés : Setosa, Versicolor et Virginica. Un ensemble de fleurs a été étudié. Pour chacune on a noté les informations suivantes : longueur et largeur des sépales, longueur et largeur des pétales.

On récupère la liste des informations sur les sépales et pétales :

```
data = sns.load_dataset('iris')
```

```
data.head()
```

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

```
D = data.values[:, :-1]
Y = data.values[:, -1]
```

```
D[:10]
```

```
array([[5.1, 3.5, 1.4, 0.2],
       [4.9, 3.0, 1.4, 0.2],
       [4.7, 3.2, 1.3, 0.2],
       [4.6, 3.1, 1.5, 0.2],
       [5.0, 3.6, 1.4, 0.2],
       [5.4, 3.9, 1.7, 0.4],
       [4.6, 3.4, 1.4, 0.3],
       [5.0, 3.4, 1.5, 0.2],
       [4.4, 2.9, 1.4, 0.2],
       [4.9, 3.1, 1.5, 0.1]], dtype=object)
```

Chaque ligne de la matrice ci-dessus est l'enregistrement des données pour une fleur.

```
Y, len(Y)
```

```
(array(['setosa', 'setosa', 'setosa', 'setosa', 'setosa', 'setosa',
       'setosa', 'setosa', 'setosa', 'setosa', 'setosa', 'setosa'],
      150)
```

```
Transform={'setosa':0,'versicolor': 1,'virginica':2}
```

```
Y=np.array([Transform[y] for y in Y])
```

```
np.array(Y)
```

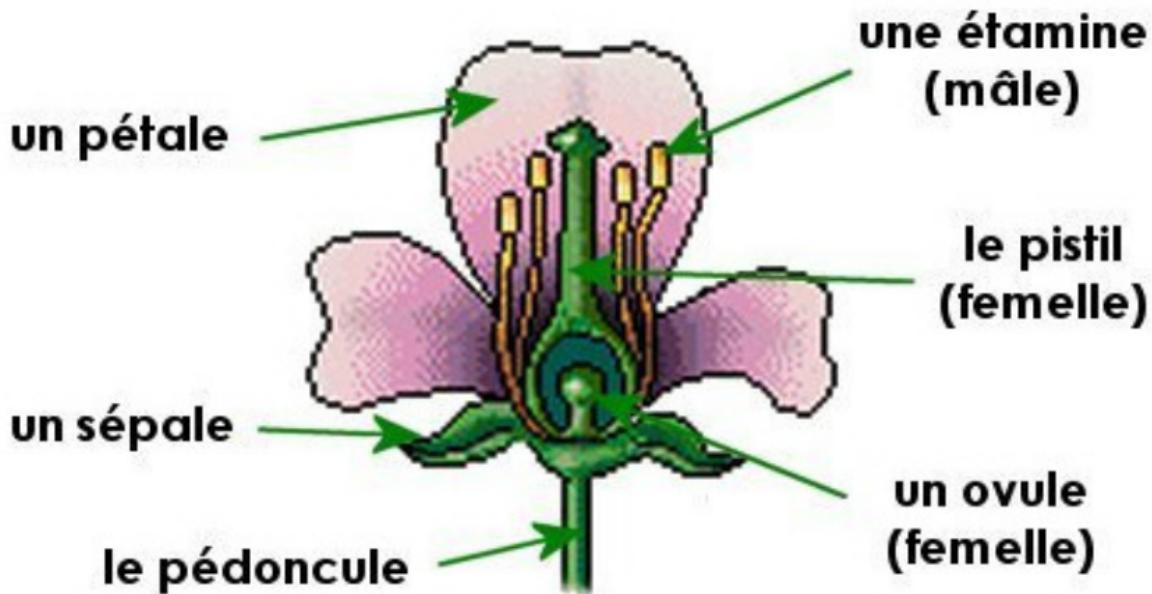
```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

Le tableau Y indique, pour chaque numéro de ligne de D la catégorie à laquelle appartient la fleur correspondante (0 pour Setosa, 1 pour Versicolor et 2 pour Virginica).

Dit autrement, le tableau Y représente une partition de l'ensemble des données. La donnée numéro i appartient à la classe d'équivalence numéro $Y[i]$.

Ainsi, $D[i]$ désigne les caractéristiques longueur et largeur des sépales de la fleur i , longueur et largeur des pétales; et $Y[i]$ représente la variété à laquelle elle appartient.

Le tableau D est appelé tableau des vecteurs caractéristiques; Y est le tableau des étiquettes de classe.



Dans tout ce qui suit D désigne une liste de coordonnées de points à classifier et Y les classes connues de ces points.

Pour fixer les idées on prendra D , Y égaux aux tableaux définis ci-dessus mais ce pourrait être tout autre chose.

8.1 Question 1

On partitionne D et Y en deux groupes selon une fonction de choix f . Pour chaque numéro de ligne de D (et donc de Y), f décide si on considère la fleur correspondante comme appartenant aux données d'apprentissage ou aux données de test. L'idéal est que f choisisse aléatoirement, mais pour contrôler nos résultats nous prenons d'abord :

```
f=lambda i:i%5!=0
```

```
def partitionner(D,Y,f):
    TestData,TestTarget,LearningData,LearningTarget=[],[],[],[]
    for i in range(len(Y)):
        if f(i):
            LearningData.append(D[i]);LearningTarget.append(Y[i])
        else:
            TestData.append(D[i]);TestTarget.append(Y[i])
    return LearningData,LearningTarget,TestData,TestTarget
```

cette fonction prend en paramètres une matrice D de données, un tableau Y de classes de ces données et renvoie 4 tableaux :

- le premier est une matrice constituée des lignes de D qui sont acceptées par la fonction f
- le second est le tableau des classes correspondants aux données acceptées

- les deux derniers tableaux correspondent aux données refusées et leurs classes

Les données acceptées (les deux premiers tableaux renvoyés) servent à l'apprentissage et les données refusées servent aux tests.

ensuite on convertit ces listes en des objets array pour faciliter leur exploitation

```
Da,Ya,Dt,Yt = convert_to_np_arrays(partitionner(D,Y,f))
```

Vous trouverez la méthode convert_to_np_arrays dans la librairie MyKNNLIB

```
print(Da[:2]) # 2 premières données d'apprentissage  
print(Dt[:2]) # 2 premières données de test
```

```
[[4.9 3.0 1.4 0.2]  
 [4.7 3.2 1.3 0.2]]  
[[5.1 3.5 1.4 0.2]  
 [5.4 3.9 1.7 0.4]]
```

Ecrire la fonction d2(x,y) qui calcule la distance entre deux points x, y de mêmes dimensions

void la définition dan dans la partie cours ou bien dans le fichier MYKNNLIB.py

```
d2(D[0],D[1])
```

```
0.5385164807134502
```

On détermine maintenant la variété de la fleur étudiée **Fleur**. On lui attribue la variété majoritaire parmi ses k plus proches voisins.

Ecrire la fonction qui prend en paramètre le dictionnaire des noms de classes, une liste de voisins d'un point et le nom d'étiquette majoritaire parmi ces voisins.

```
Fluer=[1, 4, 1, 3]
```

```
predict(Fluer, Da, Ya, d=d2, k=11)
```

```
0
```

Ainsi le type prédit de cette fleur est 0 c'est à dire le setosa

```
TestData,TestTarget,LearningData,LearningTarget=convert_to_np_arrays(partitionner(D,Y,  
 f))  
model=LearningData, LearningTarget, d1_np, 3
```

Pour tester la fiabilité du modèle on calcule la matrice de confusion dont le principe est expliqué dans la partie cours

```
matrice=ConfusionMatrix(TestData, TestTarget, model)  
print(matrice)
```

```
[[40. 0. 0.]  
 [0. 39. 1.]  
 [0. 12. 28.]]
```

BIBLIOGRAPHY

[JPBecirspahic] Jean-Pierre-Becirspahic. Arbres binaires. <https://info-l1g.fr/option-mp/pdf/01.arbres.pdf>. Accessed: 2010-09-30.