Table des matières

1 — Révisions

Dans ce chapitre, on revoit rapidement tout le cours de 1ère année, à l'exception de la partie sur les bases de données.

I Théorie des algorithmes

I.1 Notion d'algorithme

Un **algorithme** est une suite d'instructions qui
- sont exécutées dans un ordre donné sur une information connue (l'entrée de l'algorithme),
- sont non ambiguës et effectivement exécutable,
- se terminent et donnent un résultat (la sortie de l'algorithme).

Un algorithme peut s'appliquer à toute entrée admissible : on résout une classe de problème.

Du point de vue mathématique c'est une fonction : à tout entrée x de l'ensemble E (ensemble des entrées admissibles) on associe une unique image $f(x)$ (définie sans ambiguïté).

Pour valider l'algorithme, il faut donc définir l'ensemble des entrées admissibles E et montrer que pour tout x de E :

- la terminaison de l'algorithme : l'algorithme s'arrête.
- la correction de l'algorithme : la sortie est bien le résultat attendu.
- Il faut aussi calculer la complexité de l'algorithme : le temps de calcul

Il y a donc besoin de **preuves formelles** pour valider ces algorithmes.

R On ne demande pas la même rigueur qu'en mathématiques, mais de la précision.

I.2 Preuves formelles sur les algorithmes

⋆ La terminaison

On souhaite prouver que **l'algorithme s'arrête en un temps fini pour toute entrée admissible**.

- Pour les opérations simples, c'est évident.
- Les boucles `for` s'arrêtent **par définition.**
- Le problème est donc dans les boucles conditionnelles `while`

> Pour montrer la terminaison : on identifie une quantitée (variable ou expression) à valeur dans $\mathbb{N}$ qui décroît strictement à chaque itération. On parle de **variant** ou de **convergent**.
>
> Cela permet ainsi de conclure que **la boucle se termine** (puisqu'il n'existe pas de suite infinie d'entiers positifs strictement décroissante)

On peut adapter avec, par exemple, une suite strictement croissante d'entiers qui ne dépasse pas une valeur donnée, ou une suite de réels qui diminue à chaque itération d'une valeur $\varepsilon > 0$. D'une manière générale, il faut prouver qu'à un moment la condition d'arrêt du `while` va être fausse.

R L'autre difficulté que l'on verra en deuxième année est celle des algorithmes récursifs.

⋆ La correction

On veut prouver que l'algorithme **donne le bon résultat**.

> Pour montrer la correction, on utilise souvent un invariant de boucle : c'est **une propriété qui reste vraie à une ligne donnée tout au long de l'exécution de l'algorithme**.

C'est assez proche du raisonnement par récurrence : la propriété est **vraie** au premier passage (à la position considérée), si elle est vraie à un passage, alors elle reste vraie au passage suivant (**hérédité**).

Elle est alors vraie au dernier passage, ce qui prouve la correction.

R Un algorithme numérique ne donne jamais un résultat exact du fait des erreurs d'arrondi. **On a négligé ici les erreurs d'approximation**.

Pour rédiger, il faut trouver l'invariant (être capable de formaliser). La rédaction de la récurrence se fait sans difficulté particulière. **On ne demande pas de détailler**. Par contre, il est important de conclure en expliquant comment l'existence de cet invariant montre que le résultat renvoyé est correct.

⋆ La compléxité

On souhaite mesurer le temps d'exécution d'un programme, en fonction de la **taille de la donnée** en entrée, généralement mesurée sous la forme d'un entier n. Les algorithmes qui nécessitent trop d'opérations ne peuvent pas être utilisé en pratique

On calcule l'ordre de grandeur du nombre d'opérations lorsque n est grand.

- $O(1)$: le nombre d'opérations ne dépends pas de n,
- $O(n)$: de l'ordre de n.

- $O(\log(n))$, $O(n^2)$, $O(2^n)$, etc.

R Le fait de savoir que des algorithmes ont une forte complexité sert pour la sécurité (codage des mots de passe).

Pour calculer la complexité, il faut définir une unité de coût : c'est à dire l'opération qui coûte 1 temps. À savoir :

- l'affectation de variable, l'échange de deux éléments dans une liste, la comparaison de deux variables est beaucoup plus rapide que les opérations arithmétiques.
- L'addition est plus rapide que la multiplication, elle-même plus rapide que le calcul d'un cosinus, ou d'une racine carrée.
- Cela dépends de la taille et du type des nombres (multiplier par 2 est plus simple que multiplier par un grand nombre).

On peut parfois compter le nombre d'opérations arithmétiques et le nombre de comparaisons. Il arrive aussi que l'on compte le nombre d'appels à une fonction donné (ex : algorithme de dichotomie)

En pratique :
- L'énoncé indiquera quelles opérations vous devez considérer comme unité de coût.
- En l'absence d'indication, il faut compter tous les $+, \times, /$ avec un coût unitaire. Pour les algorithmes de tri (sans opérations donc), on compte le nombre de comparaisons.
- Les opérations hors des boucles **n'ont pas d'importance**.
- Les opérations dans les boucles sont répétées autant de fois que la boucle est parcourue. Il faut donc multiplier leur coût par le nombre de parcours :
  - Pour une boucle `for` c'est facile.
  - Pour une boucle `while`, il faut estimer le **nombre minimal** (meilleur cas) et **maximal** (pire des cas) de parcours.
  - Idem pour une instruction conditionnelle `if`.
- On néglige les opérations nécessaires à Python pour utiliser des boucles `for`.

I.3 Exemples d'étude théorique d'algorithme

■ **Exemple I.1** Algorithme de somme :

```
1 def somme(L):
    res = 0
3   for i in range(len(L)) :
      res +=  L[i] # invariant ici
5   return res
```

Pas de problème pour la terminaison : c'est une boucle `for` l'algorithme se termine donc.

L'invariant de boucle est :

$$\texttt{res} = \sum_{k=0}^{i} \texttt{L}[\texttt{k}]$$

On voit bien que cela est vrai au premier passage et que c'est récursif. Après la boucle, on a donc :

$$\texttt{res} = \sum_{k=0}^{n-1} \texttt{L[k]}$$

L'algorithme est donc correct.

Pour la complexité, chaque passage compte une addition, on fait n passages, on a donc $\underset{n\infty}{O}(n)$ opérations. ■

■ **Exemple I.2** Recherche du maximum

```
def calculeMax(L):
  Max = L[0]
  for i in range(len(L)) :
    if L[i] > Max :
      Max = L[i]
    # invariant ici
  return Max
```

Pas de problème pour la terminaison : c'est une boucle `for` l'algorithme se termine donc. L'invariant de boucle est :

$$\texttt{Max} = \max(L_0, \ldots, L_i)$$

Au dernier passage `Max` aura donc bien la valeur du maximum des valeurs de L. L'algorthme est donc correct.

Pour la complexité, on compte le nombre de comparaisons : `L[i] > Max`. On voit donc que l'algorithme est en $\underset{n\infty}{O}(n)$ opérations. ■

■ **Exemple I.3** Calcul de la factorielle.

```
def fact(n):
  res = 1
  while n >0 :
    # invariant ici
    res *= n
    n -= 1
  return res
```

Pour la terminaison : on a une boucle `while` mais la valeur de n diminue de 1 à chaque itération, tout en restant positive. On fait donc n passages dans la boucle. L'algorithme se termine donc.

Pour la correction : on peut écrire :

$$\texttt{res} = \prod_{i=n+1}^{n_0} i$$

avec n_0 est la valeur de n en entrée et n la valeur courante. Lorsqu'on sort de la boucle, $n = 0$ donc :

$$\texttt{res} = \prod_{i=1}^{n_0} i = n_0!$$

L'algorithme est correct.

Pour la complexité, on peut compter le nombre de multiplication : il y en a une par passage dans la boucle, donc $\underset{n\infty}{O}(n)$ multiplications au final. ■

■ **Exemple I.4** Division euclidienne par soustractions.

```
def diviEuclid(n, d):
  q = 0
  r = n
  while r >= d :
    # invariant ici
    q += 1
    r -= d
  return q, r
```

Pour la terminaison, on a une boucle `while` qui s'exécute tant que $r \geqslant d$, or la valeur de d est contante, et r diminue de d à chaque itération. L'algorithme se termine donc.

Pour la correction, on a toujours :

$$n = qd + r$$

C'est vrai à la première itération et c'est récursif car r est diminué de d tandis que q est augmenté de 1. Ce sera donc vrai après la boucle, on aura ainsi en sortie de l'algorithme :

$$n = qd + r \qquad \text{et} \qquad r < d$$

L'algorithme est correct.

Pour la complexité, on compte le nombre d'additions soustractions, il y en a 2 par passage et il y a $q = \left\lfloor \frac{n}{d} \right\rfloor$ passages. Ainsi la complexité est en $O\left(2\left\lfloor \frac{n}{d} \right\rfloor\right)$. ■

■ **Exemple I.5** Recherche d'un élément nul

```
def contient0(L):
  trouve = False
  i = 0
  while i < len(L) and not trouve :
    if L[i] == 0 :
      trouve = True
    else :
      i += 1
    # invariant ici
  return trouve
```

Pour la terminaison, on a une boucle `while`. Si `trouve` passe à `True`, alors la boucle s'arrête. Si `trouve` est toujours égal à `False` alors l'entier i augmente de 1 à chaque itération, donc i finira par dépasser la valeur de `len(L)` (qui est finie). L'algorithme va ainsi s'arrêter dans tous les cas.

Pour la correction, on a l'invariant : « `trouve` est `False` » ou » `L[i] != 0` lorsqu'on sort de la boucle, c'est :

- soit que `trouve` est `True` et `L[i]` vaut 0.

- soit que `trouve` est resté à `False` et que pour toutes les valeurs de i `L[i]` est non nul.

L'algorithme est correct.

Pour la complexité, on doit calculer le nombre de comparaison `L[i] == 0`. Dans le pire des cas (la liste ne contient pas 0), on en a n. Dans le meilleur cas (la liste commence par 0), on en a 1.

R Pensez à bien indiquer à quelle situation corresponds le meilleur / pire des cas. Ici on pourrait supposer qu'il n'y a qu'un 0 dans la liste, que sa place est uniformément répartie dans la liste et donc calculer la complexité moyenne.

■

■ **Exemple I.6** Algorithme de tri par sélection

```
def tri(L):
  n = len(L)
  for i in range(n-1) :
    # recherche du min dans L[i:n]
    indMin = i
    for j  in range(i+1, n):
      if L[j] < L[indMin] :
        indMin = j

    # on place le minimum en position i
    L[indMin], L[i] = L[i], L[indMin]

  return L
```

Pour la terminaison, pas de difficulté, il s'agit de deux boucles `for`

Pour la correction, on exécute sur un exemple et on voit qu'à la fin de la boucle `for` sur `i` :

$$L_0 \leqslant L_1 \leqslant L_2 \leqslant \ldots \leqslant L_i \leqslant \min(L_{i+1}, \ldots, L_{n-1})$$

R Cela suppose que l'on a déjà démontré la correction d'une partie de l'algorithme : la recherche du minimum.

Lorsque $i = n-1$, on obtient :

$$L_0 \leqslant L_1 \leqslant L_2 \leqslant \ldots \leqslant L_{n-1}$$

l'algorithme est donc correct.

Pour la complexité, on compte le nombre de comparaisons : `L[j] < L[indMin]`.

On a : $\sum_{i=1}^{n}(n-i-1) = \underset{n\infty}{O}\left(\frac{n^2}{2}\right)$ comparaisons. ■

■ **Exemple I.7** Algorithme de tri à bulles

```
def tri(L) :
  n = len(L)
  for j in range(n-1) : # j = numéro du passage
```

```
    for i in range(n-j-1) :
      if L[i+1] < L[i] :
        # deux éléments consécutifs dans le mauvais ordre
        L[i], L[i+1] = L[i+1], L[i]
    #invariant ici
  return L
```

Pour la terminaison, pas de difficulté, il s'agit d'une boucle `for`.

Pour la correction, on peut voir qu'à a fin de la boucle sur j, on a :

$$\max(L[0], L[1], \ldots, L[n-j-1]) \leqslant L[n-j] \leqslant \ldots \leqslant L[n-2] \leqslant L[n-1]$$

autrement dit les j derniers sont bien placés.

Pour la complexité, on a de nouveau un $\underset{n\infty}{O}\left(\frac{n^2}{2}\right)$. ∎

Exercice 1 Pour finir sur cet exemple, on peut aussi l'écrire ainsi :

```
def tri(L) :
  n = len(L)
  for j in range(n-1) : # j = numéro du passage
    modif = False # passe à True si chgt
    for i in range(n-j-1) :
      if L[i+1] < L[i] :
        # deux éléments consécutifs dans le mauvais ordre
        L[i], L[i+1] = L[i+1], L[i]
        modif = True
    if not(modif) : # pas de chgt dernier passage
      return L # fin de la fct
  return L
```

Montrer alors la correction.

Indication : montrer que si il n'y a pas de changement au dernier passage la liste est triée.

Montrer la complexité :

- Dans le pire des cas, on a toujours $\underset{n\infty}{O}\left(\frac{n^2}{2}\right)$ comparaisons. C'est le cas d'une liste de départ classé par ordre décroissant.
- Dans le meilleur des cas, on a un seul passage donc n comparaisons. C'est le cas où la liste était déjà triée.

II Fonctionnement de Python

II.1 Architecture des ordinateurs

On a essentiellement deux composants de l'ordinateur qui nous intéressent : la RAM et le processeur.

Le **processeur** est le cœur du système. Il permet **d'effectuer des calculs** :

- il va lire dans la RAM deux nombres et l'opération demandée,
- Il stocke ces informations dans ces **registres**.
- Il utilise son unité de calcul pour effectuer l'opération

- écrit le résultat dans la RAM à un endroit donné.
- Il passe ensuite à l'opération suivante

Le processeur peut aussi **se déplacer dans les instructions** :

- Il peut effectuer des opérations logiques (ex : comparer deux nombres), stocker le résultat dans ses registres.
- Un registre spécial contient la position où le processeur va lire les instructions. Le processeur peut changer cette valeur et donc **contrôler le flux d'exécution** en changeant la position de l'instruction suivante.
- Cela permet d'effectuer des branchements conditionnels `if` et des boucles `for` et `while`.

La RAM (random access memory) est la mémoire de travail de l'ordinateur :

- La RAM contient les données et les instructions nécessaires à l'exécution des programmes.
- Cette mémoire est **inerte** : aucun calcul n'est effectué par la RAM. C'est le processeur qui modifie la RAM.
- Si on éteint l'ordinateur, cette mémoire est perdue.
- La RAM est organisée de manière linéaire : c'est une suite de 0 et de 1 (sans signification a priori). On se repère sur cette ligne en donnant **l'adresse mémoire** correspondant à une position.
- On accède à toutes les adresses mémoires à la même vitesse. On peut donc fragmenter sans difficulté l'information sur des adresses très éloignées.

II.2 Architecture logicielle

Sans rentrer dans les détails, il faut rappeller l'existence du **système d'exploitation** (OS) qui est (schématiquement) le programme qui démarre lorsqu'on allume l'ordinateur et qui s'arrête lorsqu'on l'éteint. Tous les programmes doivent s'adresser à lui pour avoir accès aux ressources (en particulier pouvoir utiliser le processeur et la mémoire).

II.3 Variables et objets en Python

Pour programmer en Python, il faut avoir une connaissance (même partielle et schématique) de la manière dont Python organise la mémoire. En effet, cela a plusieurs conséquences sur la manière dont on programme.

Il faut en particulier comprendre la principale difficulté : Python doit gérer la place que lui donne l'OS dans la RAM de manière optimale. Le problème est que :

- cette place donnée par l'OS n'est pas nécessairement faite d'un seul bloc,
- les objets stockés peuvent changer de taille.
- On détruit et on crée souvent des objets.
- Il faut gérer l'information sur plusieurs niveaux : chaque fonction doit pouvoir travailler avec ses variables.

Il est facile de fragmenter l'information dans la RAM (puisque toutes les adresses sont accessibles à la même vitesse), mais il est compliqué de recopier de l'information pour la déplacer (cela demande des opérations de calcul au processeur).

La méthode utilisée par Python consiste à séparer :

- un **espace des noms** qui est la liste des noms des variables et leur adresse,

- **les objets eux même** stockés ailleurs là où l'OS a donné de la place (espace des objets).

Un objet est une **partie de la RAM qui stocke de l'information structurée**.
Un objet est défini par :
- le type de donnée stockée,
- le nombre de références à cet objet,
- la valeur de l'objet écrite en binaire.

Une variable est une **référence vers un objet**. Une variable est définie par Elle est définie par :
- le nom utilisé pour la désigner (identificateur),
- l'adresse mémoire de l'objet qu'elle référence.

La méthode utilisée par Python est spécifique à Python. D'autres langages ont fait d'autre choix.

Pour créer une nouvelle variable, par exemple le résultat d'un calcul, Python crée un nouvel objet dans la RAM et ajoute une ligne à l'espace des nom.

Pour connaître la valeur contenu dans la variable *a*, Python commence par lire son adresse dans la table des noms. Il va alors à cette adresse lire le type et, avec ce type, convertir la suite de 0 et de 1 en une valeur.

Pour détruire une variable, Python enlève une ligne à l'espace des noms. Il enlève aussi une référence à l'objet concerné. Si celui-ci n'a plus de référence, il libère l'espace mémoire.

Les instructions en Python peuvent alors :
- **créer un nouvel objet** : une nouvelle position est utilisée dans la RAM.
  - C'est l'instruction variable=expression.
  - **L'expression est résolue** (valeur et type de donnée), un nouvel objet est crée
  - `variable` est créé (ou modifiée) pour **référencer cet objet**.
- créer une **référence** : une nouvelle variable est crée dont l'adresse mémoire et le type sont identiques à une autre variable.
  - C'est l'instruction variable1=variable2.
  - La `variable1` est alors une **référence vers la `variable2`.** Elles ont la même adresse mémoire et le même type, elles ont la même valeur. On parle de référence partagée.
- **modifier un objet existant** : la RAM est modifiée à une position donnée par l'adresse mémoire.
  - C'est l'instruction variable.methode() (application d'une méthode sur l'objet), ainsi que **l'incrémentation** += et **les affectations augmentées**.
  - Si l'objet est mutable : il peut subir des modifications sans changer d'adresse mémoire. Lorsqu'on modifie cet objet, toutes les variables qui référencent cet objet voient leur valeur modifiée.
  - Pour un type **non mutable** (**immuables**), toute modification entraîne la

création d'un nouvel objet et donc **les références partagées ne sont pas modifiées**.

Les `float`, `str` et `int` sont non mutables, les `list` et `array` sont mutables.

■ **Exemple II.1** Les instructions : `a=2.7;b=2.7` et `a=2.7;b=a` sont très différentes !

Dans le premier cas : **deux objets sont crées** et le décimal 2.7 est **converti deux fois en binaire**. Dans le deuxième cas, un seul objet est créé et il n'y a qu'une conversion. ■

■ **Exemple II.2** Les instructions : `a=b` et `a=b+0` ne font pas la même chose !

Dans le premier cas, *a* est **un alias** pour *b*, dans le deuxième cas, c'est le résultat d'un calcul et c'est un objet différent. ■

■ **Exemple II.3** Les instructions :

```
a = 3.17
b = a
L = [3, 4, 5]
M = L
msg ="bonjour"
msg2 = msg
```

L'état (espace des noms et valeurs) est :

nom	type	adresse	valeur
a	float	140548538712376	3.17
b	float	140548538712376	3.17
L	list	140548537533640	[3,4,5]
M	list	140548537533640	[3,4,5]
msg	str	140548537462600	"bonjour"
msg2	str	140548537462600	"bonjour"

R L'adresse est obtenue avec la fonction `id`, le type avec la fonction `type`.

Si ensuite on utilise les instructions :

```
b += 3
M += [6]
msg += "!"
```

L'état devient :

nom	type	adresse	valeur
a	float	140548538712376	3.17
b	float	140548538714896	6.17
L	list	140548537533640	[3,4,5,6]
M	list	140548537533640	[3,4,5,6]
msg	str	140548537462600	"bonjour"
msg2	str	140548472182704	"bonjour!"

■

On a donc un comportement différent de l'incrémentation `+=` selon le type.

- pour les types non mutables (`int`, `float`, `str`), **cela crée un nouvel objet**,
- pour les types mutables (`list`), l'objet est modifiée et donc toutes ses références partagées sont modifiés.

C'est le cas pour toutes les instructions du type méthode appliquée à un objet. Le principe est que les **« gros » objets que l'on modifie souvent sont mutables**, parce qu'il est plus simple de les modifier sur place. Au contraire, les « petits » objets ou ceux qui sont peu modifiés sont non mutables.

En pratique : lorsqu'on modifie un objet mutable par une méthode, on doit garder en tête que toutes les références partagées de cet objet sont aussi modifiées.

! Attention au `L += [valeur]` qui « cache » la méthode `L.append(valeur)`.

R On parle parfois d'égalité physique (même place dans la RAM) et égalité de valeurs. L'égalité physique impliquant l'égalité de valeurs. Il y a référence croisée lorsque deux variables sont physiquement égale : non seulement les objets ont la même valeur mais les objets sont stockées exactement au même endroit. Lorsqu'il y a référence croisée sur un objet mutable, toute modification de l'objet modifie toutes les références croisées.

✭ Stockage des listes

Les listes sont des **conteneurs d'objets** de type quelconque et de taille quelconque. Stocker une liste dans la RAM est donc compliqué, car **la taille de la liste peut changer ainsi que la taille des objets qu'elle contient**.

Pour stocker des listes, Python crée un objet liste. Pour sa valeur, il alloue une place fixée dans la RAM (un certain nombre de « cases »). Dans cette place (dans ces « cases »), il stocke les références vers les objets contenus dans la liste.

Les listes contiennent ainsi des **références vers des objets** et non **les valeurs des objets**. Ainsi, si l'un des éléments de la liste est modifié et / ou change de taille, il n'a pas besoin de recopier toute la liste. Comme la « case » contient la référence vers l'objet, sa taille ne dépends pas de la taille de l'objet.

Pour connaître la valeur de `L[k]` :

- Python lit l'adresse de L (notée adL) dans l'espace des noms, il vérifie que k est bien un indice (inférieur strictement à la longueur de la liste),
- Il calcule $a = adL + k \times B$, où B est le nombre de bits nécessaire pour stocker une adresse,
- Il va lire dans la RAM à l'adresse a une adresse mémoire noté b.
- la valeur de $L[k]$ est lu à l'adresse b.

Lorsque Python doit ajouter un objet à la liste, il utilise la case suivante. Si il manque de places, Python **en alloue d'autres**.

En conséquence :

- On accède (en lecture et en écriture) aussi rapidement à l'élément 0 qu'au denier élément ou à un élément central, quelque soit la taille et le contenu de la liste. Même si le type des objets stockés sont différents et même si il s'agit de liste de listes.

- Il y a une petite perte de mémoire puisque si on crée une liste avec 3 éléments, Python va quand même allouer la place de stocker N références (avec N grand qui dépend du processeur).
  Ainsi, Python n'est pas fait pour les cas où la taille mémoire est critique.
- En contrepartie, on ajoute **très facilement des objets à la fin**. Seule petite exception : lorsqu'on ajoute le $N+1$-ième élément, Python doit attendre que l'OS lui attribue de la place dans la RAM.
  Il est plus difficile d'ajouter des objets au milieu, cela nécessite de déplacer les éléments situés après. On utilise donc plus souvent `L.append(valeur)` que `L.insert(valeur, i)`.
- On supprime très facilement un objet à la fin, en effaçant une référence. Ainsi, on utilise `y=L.pop()` qui détruit le dernier élément et stocke sa valeur dans y. Cela permet d'utiliser facilement des piles en Python.
  On efface plus difficilement un objet au milieu, même si cela peut se faire avec `y=L.pop(k)`, ou avec `del(L[k])`.
- On peut **échanger la place de deux éléments très rapidement**. Il suffit d'inverser les adresses mémoires.
  Pour cela, on utilise : `L[i], L[j] =L[j], L[i]`.

D'autres conséquences du fait que l'on stocke des références et non des objets :

- Si on met une référence dans une liste, on a un **référence partagée**. donc une modification de la liste va modifier les autre références.
- Lorsqu'on parcourt une liste, on crée une variable qui est une référence à l'objet contenu dans la liste. La **modification de cette variable modifie la liste** si le type de l'objet est mutable.

■ **Exemple II.4** Les instructions :

```
M = [4,5,6]
L = [ 1 , [2,3], M]
N = L[1]
M.append(1)
print("après modif de M:")
print("M",M)
print("L",L)
L[1].append("bonjour")
print("après modif de L:")
print("N", N)
print("L",L)
```

vont donner :

```
après modif de M:
M [4, 5, 6, 1]
L [1, [2, 3], [4, 5, 6, 1]]
après modif de L:
N [2, 3, 'bonjour']
L [1, [2, 3, 'bonjour'], [4, 5, 6, 1]]
```

La liste de listes `L` a donc été modifiée par la modification de `M`, la liste `N` a été aussi modifiée par la modification de `L`. ■

■ **Exemple II.5** Les instructions :

```
listeListes = [ [], ["a"], ["b"], ["a","b"]]
for liste in listeListes :
  liste.append("c")
```

donnent :

```
[['c'],['a','c'],['b','c'],['a','b','c']]
```

et les instructions :

```
LL = [ [0], [1,2], [3,4,5], [6,7,8,9] ]
for L in LL :
  L[-1] = 3
```

donnent :

```
[[3], [1, 3], [3, 4, 3], [6, 7, 8, 3]]
```

Dans les deux cas, la liste de liste a été modifiée dans la boucle `for`. ■

■ **Exemple II.6 Parcours de liste de liste** On peut parcourir une liste de liste. La variable de boucle prends alors pour valeur chacune des listes de la liste de listes.

Par exemple, pour générer les parties de $\{a,b,c\}$ à partir des parties de $\{a,b\}$

```
partieAB = [ [], ['a'], ['b'], ['a','b'] ]
partieABC = []
for partie in partieAB :
    partieABC.append(partie)
    partieABC.append(partie + ['c'])
```

Cette fonctionnalité est très utile pour les algorithmes de dénombrements ! ■

✯ Espace des noms des fonctions

Lorsqu'on écrit une fonction, on écrit d'un algorithme : on écrit des instructions qui permettent à partir d'une entrée x de calculer une sortie y. On essaie au maximum de détacher cette partie du programme du reste du code : la fonction est faite pour être appelée sur des entrées différentes et sa sortie est faite pour être utilisée de différentes manières. L'idée est aussi qu'une fonction doit pouvoir être utilisée par un autre programmeur : celui-ci a juste accès à l'entrée et la sortie de la fonction et à un descriptif de ce qu'elle fait.

Le nom que l'on donne aux variables x (pour l'entrée), y (pour la sortie) ainsi que toutes les variables nécessaires pour effectuer ces instructions ne doit donc pas se confondre avec les noms utilisées dans l'espace appelant la fonction ou dans d'autres fonctions.

Une fonction a ainsi son **propre espace de noms**, les références crées par la fonction ne peuvent jamais être utilisé en dehors de la fonction. Lorsqu'on sort de la fonction (par `return`), **tous les objets crées par la fonction sont détruits**.

> En pratique :
> - Si une fonction utilise une variable `n`, alors en dehors de la fonction, on ne pourra pas connaître la valeur de n. **Celle-ci n'est pas dans l'espace de noms du script**.

- Si en dehors de la fonction une autre variable `n` existe, alors il n'y a pas de collision. Les deux variables sont dans des espaces de noms différents.

■ **Exemple II.7** On crée une fonction qui utilise une variable *n*

```
def f():
    n = 2
    print("dans fonction")
    print("n", n, type(n), id(n))
f()
print("dans script")
print("n", n, type(n), id(n))
```

La variable `n` n'existe pas ailleurs que dans la fonction.

```
dans fonction
n 2 <class 'int'> 10105856
dans script
NameError: name 'n' is not defined
```

■

■ **Exemple II.8** On crée une fonction qui utilise une variable *n*, alors qu'une autre variable *n* existe dans le script.

```
def f():
    n = 2
    print("dans fonction")
    print("n", n, type(n), id(n))
n=3.5
f()
print("dans script")
print("n", n, type(n), id(n))
```

Chaque variable est associée à son propre espace de noms. Il n'y a pas de collision.

```
dans fonction
n 2 <class 'int'> 10105856
dans script
n 3.5 <class 'float'> 140489389161232
```

Les deux variables font références à des objets différents (et n'ont pas la même valeur). ■

Lorsque Python doit **évaluer une expression** dans une fonction, il remplace les variables par les valeurs référencées. Par quelle valeur remplacer la variable ?

Python utilise la règle dite `LEG` :

- Il regarde d'abords **localement** (ie dans la fonction),
- si il ne trouve pas, il regarde dans **l'espace de nom englobant** (si la fonction est dans une fonction).
- sinon il regarde dans **l'espace de noms global** (ie le script).

Ainsi, une variable **définie dans le script pourra être utilisée dans une fonction**

Pour éviter les confusions, si une variable `x` est utilisée ainsi, alors le nom `x` est interdit dans **l'espace des noms de la fonction**.

On peut dire que les variables externes à la fonction sont accessibles en lecture mais pas en écriture (c'est valable pour les type immuables).

On se sert souvent de cette fonctionnalité pour utiliser des **constantes nommées** : il s'agit de variables qui ne changent pas de valeur et qui sont accessibles dans la fonction. Par exemple, c'est pratique pour les noms de fichiers, les constantes physique, etc. Cela évite d'avoir beaucoup d'entrées à la fonction.

Un autre intérêt de cette pratique est de pouvoir utiliser des fonctions dans d'autres fonctions. Si on crée une fonction f, alors le nom f dans l'espace des noms du script existe (et désigne l'objet fonction). Si on crée une autre fonction g et que dans g on utilise la fonction f, alors l'interpréteur va chercher dans l'espace appelant (ici le script) la variable f. On peut donc utiliser la fonction f dans la fonction g. Notons que si on utilise une fois la fonction f dans g alors le nom f ne peut pas désigner autre chose.

■ **Exemple II.9** La variable n existe dans le script, on essaie de l'utiliser dans la fonction.

```
def f():
    print("dans fonction")
    print("n", n, type(n), id(n))
n=3.5
f()
print("en dehors de la fonction")
print("n", n, type(n), id(n))
```

Ici, Python utilise la règle LEG : il va chercher dans l'espace des noms du script, l'objet n.

```
dans fonction
n 3.5 <class 'float'> 140423282762280
en dehors de la fonction
n 3.5 <class 'float'> 140423282762280
```

La référence n est ainsi accessible dans la fonction

■

■ **Exemple II.10** La variable n existe dans le script, on essaie de la modifier dans la fonction.

```
def f():
    print("dans fonction")
    print("n", n, type(n), id(n))
    n = 5 # instruction après utilisation
n=3.5
f()
print("en dehors de la fonction")
print("n", n, type(n), id(n))
```

Cela donne :

```
dans fonction
UnboundLocalError:
local variable 'n'
```

```
referenced before assignment
```

La variable *n* de la fonction est utilisée avant d'être affectée. On voit qu'il n'y a pas de problème si la variable *n* est utilisée après avoir été affectée. Dans ce cas, c'est deux variables différentes.

On peut dire que la variable *n* est disponible en lecture pas en écriture ! Cela est bien sûr valable uniquement pour les types immuables, précisément, la variable ne peut pas être ré-affectée (mais on peut appliquer des méthodes dessus). ■

Si dans une fonction, on déclare une variable globale, alors, Python va utiliser **l'espace des noms du script pour cette variable**. La fonction pourra alors **modifier la variable du script**.

C'est considéré comme un **mauvais style de programmation** mais le programme d'IPT demande de le mentionner.

■ **Exemple II.11** Les instructions :

```
def f():
    global n
    print("dans fonction")
    n = 5
    print("n", n, type(n), id(n))
print("avant fonction")
n=3.5
print("n", n, type(n), id(n))
f()
print("après fonction")
print("n", n, type(n), id(n))
```

donnent :

```
avant fonction
n 3.5 <class 'float'> 140587605392024
dans fonction
n 5 <class 'int'> 10105952
après fonction
n 5 <class 'int'> 10105952
```

La variable *n* est ainsi modifiée par la fonction. ■

Les entrées des fonctions sont **passées par référence** à la fonction, lorsqu'on appelle la fonction, les références des entrées sont copiés dans l'espace de noms de la fonction. Il y a donc une référence partagée puisqu'un objet est référencé dans l'espace des noms de la fonction et du script.

Cela permet à Python d'éviter de recopier les variables passées en entrée à une fonction. Par exemple, si on crée une fonction qui prend en entrée une image numérique (donc une variable qui prend beaucoup de place en mémoire), Python n'a pas besoin de recopier cet objet : il fait simplement une référence partagée.

En pratique, tout dépend du type d'objet en entrée :

- Si l'objet est **immuable**, alors cela ne pose pas de problème : si la fonction

modifie l'objet en entrée, alors un nouvel objet est crée (l'ancien n'est pas détruit puisqu'il est encore référencé dans l'espace appelant).
- Si l'objet est **mutable**, alors la fonction peut modifier l'objet en entrée ! La fonction agit alors sur la valeur d'une variable extérieure au script.

■ **Exemple II.12** Prenons une fonction avec une entrée x que l'on appelle sur en prenant pour valeur de x la variable n :

```
def f(x):
    print("dans fonction")
    print("x", x, type(x), id(x))
print("dans script")
n=3.5
print("n", n, type(n), id(n))
f(n)
```

```
dans script
n 3.5 <class 'float'> 140587605392024
dans fonction
x 3.5 <class 'float'> 140587605392024
```

La variable n (dans le script) et x (dans la fonction) sont donc le même objet (égalité physique). On a donc une référence partagée. Ce qui évite à Python de recopier les objets. Ici cela n'a pas beaucoup d'intérêt mais on peut imaginer que la variable n contient une liste de milliards de valeurs. Python ne veut pas recopier cette liste.

Ici c'est un type non mutable, donc une modification de x dans la fonction va créer un nouvel objet qui ne sera donc plus l'objet en entrée. Ce n'est donc pas un problème.

■

■ **Exemple II.13** Modifions la fonction précédente pour qu'elle modifie la variable x (précisément, on applique une méthode sur la variable x)

```
def f(x):
    print("dans fonction")
    print("x", x, type(x), id(x))
    x += 1
    print("dans fonction 2")
    print("x", x, type(x), id(x))
n=3.5
print("dans script avant fonction")
print("n", n, type(n), id(n))
f(n)
print("dans script après fonction")
print("n", n, type(n), id(n))
```

Comme il s'agit de type non mutable, on va avoir :

```
dans script avant fonction
n 3.5 <class 'float'> 140587605392144
dans fonction
x 3.5 <class 'float'> 140587605392144
dans fonction 2
x 4.5 <class 'float'> 140587605392024
```

```
dans script après fonction
n 3.5 <class 'float'> 140587605392144
```

On voit que n et x sont des références partagées, mais la modification de x dans la fonction ne modifie pas n. Pour les types mutables, pas de problème : la variable n n'est pas modifiée

■

Pour des variables mutables en entrée, on peut avoir une difficulté : la fonction peut appliquer une méthode sur son entrée et donc modifier la valeur de l'entrée dans le script.

■ **Exemple II.14** La même fonction que précédemment mais sur une liste :

```
def f(x):
    print("dans fonction")
    print("x", x, type(x), id(x))
    x += [1] # ou x.append(1)
    print("dans fonction 2")
    print("x", x, type(x), id(x))
n = [2,3,4]
print("dans script avant fonction")
print("n", n, type(n), id(n))
f(n)
print("dans script après fonction")
print("n", n, type(n), id(n))
```

La variable en entrée n est une liste donc mutable. On a alors :

```
dans script avant fonction
n [2, 3, 4] <class 'list'> 140587538448072
dans fonction
x [2, 3, 4] <class 'list'> 140587538448072
dans fonction 2
x [2, 3, 4, 1] <class 'list'> 140587538448072
dans script après fonction
n [2, 3, 4, 1] <class 'list'> 140587538448072
```

On voit que la fonction f a modifié la valeur de la variable n. Une fonction peut donc agir sur ses entrées. ■

II.4 Représentation des nombres

- La **mémoire** de l'ordinateur est constituée de cellules électroniques qui ne peuvent être que dans deux états : **sous-tension** (noté 0) ou **hors tension** (noté 1).
- Ainsi, toutes les données que manipule l'ordinateur doivent être **codées sous la forme d'une suite de** 0 **et de** 1**.**
- On appelle **bit** une cellule de la mémoire qui peut donc contenir uniquement les valeurs 0 ou 1.
- Un **mot mémoire** est une suite finie de bits.
- Souvent, on regroupe les bits par 8 pour faire un **octet**.

On a donc le problème suivant : comment **représenter les nombres sous la forme de mot mémoire** et surtout quelles conséquences a cette représentation sur les algorithmes. Il faut garder en tête les deux contraintes : n'utiliser que des 0 / 1 et en nombre fini.

✯ Écriture binaire des entiers naturels

Tout entier naturel $n \in \mathbb{N}^*$ peut se décomposer de manière unique sous la forme :

$$n = \sum_{l=0}^{p} a_l 2^l \qquad \text{où } a_p \neq 0$$

C'est **l'écriture en base** 2 **de l'entier** n.

On note alors $n = \underline{a_p a_{p-1} \dots a_0}$ pour indiquer que l'écriture en base 2 de n est : $a_p a_{p-1} \dots a_0$. Ainsi : $n = \underline{a_p a_{p-1} \dots a_0}$ signifie $n = \sum_{l=0}^{p} a_l 2^l$.

Le dernier chiffre de l'écriture binaire de n donne la parité. et les bits sont rangés du **poids fort** (le coefficient devant 2^p) au **poids faible** (le coefficient devant 2^0).

■ **Exemple II.15** $\underline{1101}$ est l'entier $n = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 1^0 = 13$ ■

D'une manière générale, l'écriture d'un **entier naturel** n **en base** k consiste à trouver une suites d'entiers $(a_0, \dots, a_p)$ tels que : $n = \sum_{l=0}^{p} a_l k^p$ avec $a_p \neq 0$ et $\forall l \in [\![0, p]\!]$, $a_l \in [\![0, k-1]\!]$.

La base usuelle est la base 10. Ainsi, on écrit naturellement :

$$1456 = 1 \times 10^3 + 4 \times 10^2 + 5 \times 10^1 + 6 \times 10^0$$

Une autre base très utilisé en informatique est la base 16 (**hexadécimale**). Cela revient à grouper les mots mémoire en paquet de 4 bits, ie en entier de $[\![0, 15]\!]$. On utilise alors les chiffres de 0 à 9 puis les lettres A,B,C,D,E,F.

Étant donné un entier naturel n et un entier $k > 0$, l'écriture de n en base k s'obtient **par division euclidienne successive** et en stockant **les restes dans une liste**.

Attention à bien mettre les bits de poids forts au début.

■ **Exemple II.16** Si on veut décomposer en binaire l'entier 589, on écrit :

$$\begin{aligned} 589 &= 2 \times 294 + 1 & 294 &= 2 \times 147 + 0 \\ 147 &= 2 \times 73 + 1 & 73 &= 2 \times 36 + 1 \\ 36 &= 2 \times 18 + 0 & 18 &= 2 \times 9 + 0 \\ 9 &= 2 \times 4 + 1 & 4 &= 2 \times 2 + 0 \\ 2 &= 2 \times 1 + 0 & 1 &= 2 \times 0 + 1 \end{aligned}$$

Ce qui donne : $589 = \underline{1001001101}$

■

Le programme de décomposition d'un entier en base k est :

```
def ecritBase(n,k):
  """
  entrée: k >0 = entier = base
     n =  entier naturel = nbr à convertir
  sortie: rep = chaine de caractère
      = écriture de n dans la base k
  """
  if n==0 :
      return "0"
  rep=""
  while n != 0 :
    rep = str(n%k)+rep # ajout du reste au début de rep
    n = n //k
  return rep
```

On a choisi ici de stocker la représentation en base k sous la forme d'une chaîne de caractères.

> Souvent, on fixe la taille de la représentation d'un entier en binaire pour l'écrire dans **un mot mémoire avec une taille fixé**. On ajoute des 0 au début de la représentation en binaire si besoin.

Par exemple, le type `uint8` correspond à représenter un entier naturel (`unsigned int`) sur 8 bits (ie sur un octet). Ce type permet donc de stocker un entier de $[\![0,255]\!]$. Il est particulièrement utilisé pour stocker les images numériques. L'avantage de cette représentation est que **le processeur calcule plus vite**, mais on a un risque de dépassement de capacité : si on dépasse la taille du plus grand entier **le calcul est faux**. D'une manière générale, si l est la taille du mot mémoire, alors on ne peut représenter que les entiers de 0 à $2^l - 1$.

■ **Exemple II.17** Soit le code :

```
>>> from numpy import uint8
>>> uint8(317) # on dépasse la capacité
61
>>> a,b  = uint8(202), uint8(115)
>>> a+b # on fait un calcul qui dépasse la capacité
__main__:1: RuntimeWarning: overflow encountered
            in ubyte_scalars
61
```

On voit que $202+115=317$ ne peut être représenté par un `uint8`. Comme on a : $317-256=61$, Python le remplace par 61. ■

> ! Python affiche un message d'avertissement pour prévenir qu'il faut faire attention, ce n'est pas un message d'erreur.

En pratique :

- Le type `int` de python **n'a pas de taille maximale** : le processeur adapte la taille de l'entier. De base, un entier est stockée sur 64 bits, si il dépasse la

capacité, Python va créer un nouvel objet avec une place plus importante. Ainsi, on peut effectuer des opérations sur des entiers de taille quelconque sans limitation.
- Il peut arriver que l'on manipule des entiers de taille fixé, généralement le type `uint8`. Dans ce cas, il faut veiller à ne pas dépasser la capacité.

✯ Le cas des entiers relatifs

Pour des entiers relatifs, on ne réserve pas un bit pour le signe. Au lieu de cela, on utilise le complément à 2.

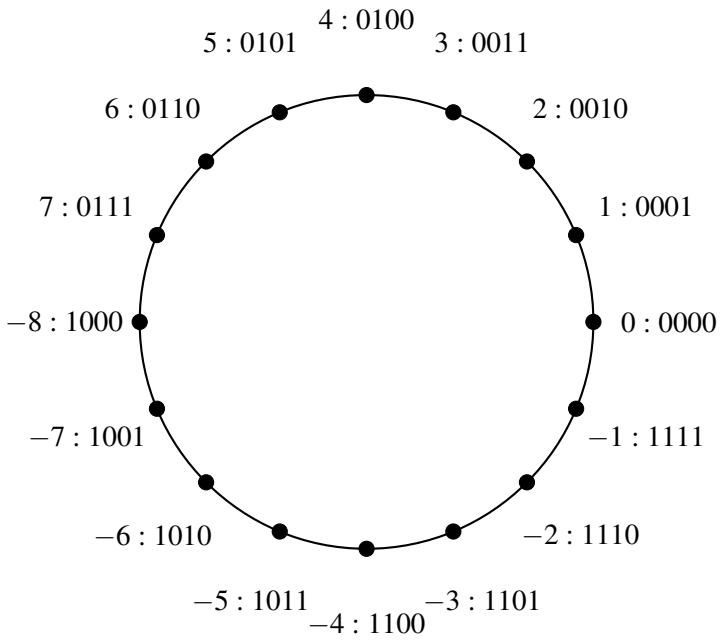
Pour représenter un entier naturel $x \in [\![-2^{n-1}, 2^{n-1}-1]\!]$ sur n bits on procède ainsi :
- Si x est positif ou nul, on le représente comme un entier naturel (sur n bits),
- si x est strictement négatif, **on le représente comme** $2^n + x$ (qui est donc un entier de $[\![2^{n-1}, 2^n - 1]\!]$).

Cela permet d'avoir une unique représentation de 0 et d'avoir une **opération d'addition naturelle**.

On représente cette méthode avec un cercle.

Prenons l'exemple où $n = 3$, on veut représenter les entiers de $[\![-8, 7]\!]$. Les nombres positifs sont représentés de la manière habituelle, -1 est représenté comme 15, -2 comme 14, etc. jusqu'à -8 qui se représente comme 8.

R On sait immédiatement le signe d'un élément cela se lit sur le premier bit.



On peut par exemple vérifier que :

$$\begin{aligned} 2+4 &: \underline{0010}+\underline{0100}=\underline{0110} : 6 \\ (-2)+(-4) &: \underline{1110}+\underline{1100}=\underline{11010} \text{ tronqué en } \underline{1010} : -6 \\ 2+(-5) &: \underline{0010}+\underline{1011}=\underline{1101} : -3 \end{aligned}$$

On voit facilement que l'opération d'addition est naturelle. On voit aussi que l'on compare très facilement deux nombres : on regarde d'abords le signe, puis on compare les bits suivants de manière naturelle.

En pratique : on retiendra le principe du complément à 2.

✯ Représentation en virgule flottante

En base 10, pour écrire un **nombre décimal** on utilise :
- Le signe $+$ ou $-$,
- la **mantisse** : un nombre décimal de $[1,10[$.
- on multiplie par 10^e où e est **l'exposant**.

■ Exemple II.18

$6.62606957 \times 10^{-34}$	constante de Planck
$6.02214129 \times 10^{23}$	Nombre d'Avogadro
$8.9875517873681764 \times 10^{9}$	Constante de Coulomb

■

On peut ainsi représenter des nombres **arbitrairement grand ou petit**. Lors des opérations d'addition ou de multiplication, on peut déterminer la précision du calcul (le nombre de **chiffres significatifs**).

Pour utiliser des nombres réels, Python fait de la même manière, sauf que tout est converti en base 2.

Pour représenter un réel en virgule flottante, on utilise **la notation signe / mantisse / exposant** en binaire.

Un réel x est ainsi écrit sous la forme : $x = (-1)^s m 2^e$
- s représente **le signe** : c'est un bit (0 ou 1). Le signe de x est donc $(-1)^s$
- e est **l'exposant**, c'est un entier relatif compris entre -1022 et 1023. On le représente sous la forme de l'entier naturel $e+1023$ qui est donc compris entre 1 et 2046 et donc codé sur 11 bits.
- m est **la mantisse** : c'est un nombre décimal écrit en binaire.
  Le seul chiffre avant la virgule est 1 et il y a 52 chiffres (0 ou 1) après la virgule.
  La mantisse est ainsi codés sur 52 bits : $m = \underline{1,a_1 \dots a_{52}}$.

Pour représenter sous forme décimale en base 10 un nombre à virgule flottante donné en binaire :
- On identifie le **signe** s (le premier bit), **l'exposant** (les 11 bits suivants) $(e_1,\dots,e_{11})$ et la **mantisse** la mantisse (les derniers 52 bits) $(m_1,\dots,m_{52})$.

- Ne pas oublier de tenir compte du décalage de 1023 pour l'exposant : $e = \underline{e_1 \dots e_{11}} - 1023$.
- Pour la mantisse, il faut calculer le nombre décimal de $[0,1[$ correspondant :

$$m = \underline{1,m_1 \dots m_{52}} = 1 + \sum_{j=1}^{52} m_j 2^{-j}.$$

- on obtient : $(-1)^s \times 2^{\underline{e_1 \dots e_{11}}-1023} \left(1 + \sum_{j=1}^{52} m_j 2^{-j}\right)$

■ **Exemple II.19** On considère :

`1 10001000110 1001001111000011110000000000000000000000000000000000000000000000.`

- Le signe est 1 ie **négatif**
- L'exposant a pour écriture binaire :

$$\underline{10001000110} = 2 + 2^2 + 2^6 + 2^{10} = 2 + 4 + 64 + 1024 = 1094$$

C'est donc $1094 - 1023 = 71$.

- La **mantisse** a pour écriture binaire décimale :

$$\underline{1.10010011110000111000000000000000000000000000000000000000000000}$$

C'est donc

$$1 + \frac{1}{2} + \frac{1}{16} + \frac{1}{2^7} + \frac{1}{2^8} + \frac{1}{2^9} + \frac{1}{2^{10}} + \frac{1}{2^{15}} + \frac{1}{2^{16}} + \frac{1}{2^{17}} = \frac{206727}{131072}$$

Au final, la valeur de x est : $x = -\frac{206727}{131072} \times 2^{71}$ ■

En pratique : ne pas trop se focaliser sur les détails techniques, il faut surtout avoir compris le principe et les conséquences de cette représentation.

Il faut bien comprendre qu'un ordinateur ne calcule qu'avec des nombres de cette forme.

Conséquences :

- Tous les décimaux ne peuvent pas s'écrire comme la somme finie :

$$(-1)^s \times 2^{\underline{e_1 \dots e_{11}}-1023} \left(1 + \sum_{j=1}^{52} m_j 2^{-j}\right)$$

Des décimaux très simples en base 10 comme 0.4 sont alors représentés par une **valeur approchée**, la plus proche possible.
Il y a donc toujours des erreurs avant même d'effectuer un calcul (erreur de représentation).

- **La taille de la mantisse est fixe** : on a donc un nombre fixé de chiffres significatifs. C'est à nous de savoir combien de chiffre sont significatifs dans le calcul effectué par la machine.
- On ne peut pas représenter des nombres flottants qui ont un exposant plus

grand que 1024. Le plus grand flottant possible est ainsi

$$1.7976931348623157 \times 10^{308} = 2^{1024}$$

Si on dépasse ce nombre, on a une erreur de **dépassement de capacité**.

- On ne peut pas représenter des nombres flottants qui ont un exposant plus petit que -1021. Le plus petit flottant possible est ainsi

$$2.2250738585072014 \times 10^{-308}$$

Un tel flottant est alors représenté par 0.

- On ne conserve que 53 décimale (en binaire) après la virgule, ainsi $1+2^{-54}$ est représenté comme 1.
On appelle ε la **précision de cette représentation**, c'est le plus petit réel tel que : $1+\varepsilon \neq 1$ Dans notre modèle ce réel est 2^{-52}. qui est de l'ordre de 10^{-16}.
On peut donc considérer que chaque calcul est fait à la **précision relative** 10^{-16}. Dans le sens où si a est un réel $a+\varepsilon a$ est remplacé par a.
Attention :cela ne signifie pas que l'on ne peut pas manipuler des nombres inférieurs à 10^{-16}. Au contraire, la représentation en virgule flottante permet de travailler avec des nombres jusqu'à 10^{-307}. Cela signifie que Python calcule toujours avec 16 chiffres significatifs.
La principale conséquence est qu'un test d'égalité == entre flottants n'a pas de sens.
Toute opération arithmétique induit aussi des erreurs relatives d'approximation de l'ordre de 10^{-16} puisqu'on ne maîtrise pas les dernières décimales.

■ **Exemple II.20** Voici un exemple de dépassement de capacité :

```
>>> a = 256
>>> a**a # le calcul se fait sans problème
>>> a=256.
>>> a**a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: (34, 'Numerical result out of range')
```

■

■ **Exemple II.21** Voici un example de flottant identifié à 0.

```
>>> a=10**(-20)
>>> a=a**2
>>> a
1e-40
>>> a=a**2
>>> a
9.999999999999998e-81
>>> a=a**2
>>> a
9.999999999999995e-161
>>> a=a**2
>>> a
```

```
1e-320
>>> a=a**2
>>> a
0.0
>>> a==0
True
```

■

■ **Exemple II.22** Voici un exemple permettant de calculer la valeur de ε.

```
>>> eps = 1.
>>> i = 0
>>> while 1+eps != 1 :
...     i += 1
...     eps = eps /10
>>> print(i)
16
>>> print(eps)
1.0000000000000001e-16
```

■

■ **Exemple II.23** Le code suivant affiche souvent un message d'erreur :

```
a = rand() # aléatoire dans [0,1[
b = rand() # idem
x = -b/a
if a*x+b == 0 :
  print("ok")
else:
  print("erreur")
```

■

■ **Exemple II.24** Cette boucle qui devrait s'arrêter est en fait une boucle infinie :

```
x = 0.
while x != 1.:
  x += 0.1
```

Par contre, la même boucle mais sur des entiers (avec un calcul exact donc), s'arrête toujours :

```
i = 0.
while  i != 10
  i += 1
```

■

⋆ Conclusion

À retenir :

- L'algorithme de **conversion d'un nombre en binaire** et dans une base quelconque.
- Le principe du complémentaire à 2.

- Le principe de la **représentation à virgule flottante.**
- Les conséquences de cette représentation :
  - les erreurs de représentation,
  - la gestion du **nombre de chiffres significatifs**,
  - les dépassements de capacités,
  - les **erreurs dans les additions,** lorsqu'on ajoute une très petite valeur et une grande,
  - toute opération entre flottants provoque une erreur,
  - le test d'égalité entre flottants n'a jamais de sens.

✯ Exemples de problème d'arrondi

Comparaison des deux formules pour la variance :

On a deux formules pour la variance :

$$\sigma^2(l) = \frac{1}{n}\sum_{k=0}^{n-1}\left(l_i - \bar{l}\right)^2 \qquad \text{et} \qquad \sigma^2(l) = \overline{l^2} - \left(\bar{l}\right)^2$$

la propagation des erreurs d'arrondis n'est pas la même :

```
n = 1000
l = [ float(10**7 + (-1)**i) for i in range(n) ]
# série statistique égale à 10**7 quasi constante

# calcul de l'espérance:
m = sum(l)/n
print(m) # donne 10000000.0 soit 10**7

# calcul de la variance par la définition
var1 = sum([(x-m)**2 for x in l])/n
print(var1) # donne 1.0

# calcul de la variance par la formule de Koenig-Huygens.
var2 = sum([x**2  for x in l])/n - m**2
print(var2) # donne 0.09375
```

La formule

$$\sigma^2(l) = \frac{1}{n}\sum_{k=0}^{n-1}\left(l_i - \bar{l}\right)^2$$

est donc correcte, tandis que la formule

$$\sigma^2(l) = \overline{l^2} - \left(\bar{l}\right)^2$$

est grossièrement fausse !

Cela provient des erreurs d'arrondis :

- Ici $l_i = 10^7 + (-1)^i$ et donc $\bar{l} = 10^7$ et $\sigma^2(l) = 1$.
- On a alors : $\left(l_i - \bar{l}\right)^2 = 1$ donc $\frac{1}{n}\sum_{k=0}^{n-1}\left(l_i - \bar{l}\right)^2 = 1$
- Par contre : $l_i^2 = 10^{14} + 2 \times 10^7 \times (-1)^i + 1$

- lorsque l'on calcule $\sum_{i=1}^{n} l_i^2$, au bout de quelques itérations, la somme dépasse 10^{15} et le 1 est négligé.
- Tout se passe alors comme si : $\sum_{i=1}^{n} l_i^2 \approx 10^{14} \times n$ D'où $\overline{l^2} - \left(\overline{l}\right)^2 \approx 0$!

La série harmonique

On considère $S_n = \sum_{k=1}^{n} \frac{1}{k}$. Une étude mathématique classique montre que S_n est strictement croissante et tends vers $+\infty$. En fait $S_n \underset{+\infty}{\sim} \ln(n)$.

On ajoute 10^{12} pour voir les erreurs d'arrondi :

```
s1 = sum ([ 10**12] + [1/k for k  in range(1, 10**5)])
s2 = sum ([ 10**12] + [1/k for k  in range(1, 10**6)])
print(s1, s2, s1==s2)
```

donne :

```
1000000000010.5522 1000000000010.5522 True
```

La suite semble donc converger !

R En pratique, lorsqu'on calcule la somme des termes d'une somme, il est plus précis d'ajouter les éléments « par la fin », ie du plus petit au plus grand.

Une suite particulière

Exercice 2 Soit la suite (u_n) définie par :

$$u_0 = 6, \quad u_1 = 6, \quad \forall n \in \mathbb{N},\ u_{n+1} = 111 - \frac{1130}{u_n} + \frac{3000}{u_n \times u_{n-1}}$$

1. Calculer les 50 premiers termes de la suite.
2. Quel semble être le comportement de la suite ?
3. Montrer par récurrence double que la suite est constante.

Correction :

1. Voici un exemple de code :

```
1
NBRTERME = 50
3 u = [0]* NBRTERME
u[0] = 6
5 u[1] = 6
for n in range(1,NBRTERME-1):
7     u[n+1] = 111 - 1130/u[n] +  3000/(u[n]*u[n-1])

9 print("--- suite des termes ---")
print(u)
11 print("--- les 10 derniers ---")
print(u[-10:])
```

2. La suite semble converger vers 100.
3. On a :

$$111 - \frac{1130}{6} + \frac{3000}{6^2} = 6$$

Donc par récurrence double immédiate, la suite est constante égale à 6.

III Programmer en Python

Programmer dans un language c'est connaître les fondamentaux suivants :

- Les **variables** : stocker des valeurs et les manipuler,
- Les **boucles** : Blocs d'instructions répétées
  - un certain nombre de fois connu à l'avance (`for`),
  - tant qu'une condition est vérifiée (`while`).
- Les **conditions** : exécuter un bloc ou un autre selon une condition (`if`)
- Les **fonctions** : bloc d'instructions qui peuvent être utilisés plusieurs fois sur des entrées différentes.
- les **conteneurs** : regrouper des valeurs dans des listes (ou des tableaux) et les manipuler.

Pour utiliser L'éditeur de développement intégré Spyder, il est fortement conseillé d'utiliser les options d'exécution par le menu
`Exécution -> configurer` (`[CTRL+F6]` ou `[F6]`).

- Exécuter dans un nouvel interpréteur Python dédié ou Exécuter dans un terminal système externe.
- option Interagir avec l'interpréteur Python après l'exécution.

Un raccourci très utile est la complétion `CTRL+ESPACE`.

Pas d'accent, pas d'espace dans les noms de fichiers, ni dans les noms de dossier ! De même, pas d'accent dans les noms de variables (même si cela est licite en Python3).

Autre idée importante : la lisibilité compte ! Il faut être capable d'écrire du code qui sera lu et compris rapidement par d'autres.

Pour rendre le code plus lisible, il faut utiliser des noms de variables qui ont un sens et organiser le code. On peut mettre une instruction sur plusieurs lignes avec des parenthèses.

■ **Exemple III.1** Voici un exemple de code tiré d'un projet : I, J, K, L, M et N sont des points du plan.

On a une partie du plan qui nous intéresse, et on crée donc une fonction qui à un point (x, y) renvoie `True` ou `False` pour indiquer si le point est dans la bonne zone.

Pour chaque point (x, y) on a calculé des booléens indiquant la position du point (x, y) par rapport aux différentes droites.

On écrit :

```
return (  (sousIJ and surML and gaucheJM)
       or (sousJK and surMN and droiteJM) )
```

Le code est ainsi rendu très lisible. ■

■ **Exemple III.2** On veut disposer des cercles aléatoirement dans une image, **sans qu'ils se touchent**. Une partie de l'algorithme peut s'écrire ainsi : on a une liste de cercles `Lcercle` et on veut ajouter un cercle au hasard sans qu'il touche ceux dans la liste.

Voici une partie du code :

```
convient = False # passe à True si ok
while not(convient):
  candidat = unCercleHasard()
  convient = True # ne stoppe pas la boucle
  for cercle in Lcercle :
    if touche(cercle, candidat) :
      convient = False
Lcercle.append(candidat) # ajoute candidat à Lcercle
```

On utilise volontairement des fonctions pour « découper le code » et le rendre plus lisible. ■

✯ Accès à l'aide

En pratique, on utilise souvent internet pour chercher de l'aide et l'explorateur de variable graphique de l'IDE pour décrire les variables et leur valeurs. Néanmoins, il faut savoir faire sans, en particulier pour le jour de l'oral (pas d'accès au net).

Pour avoir accès à l'aide, on peut utiliser :

- `dir()` donne la liste des bibliothèques chargées et les variables affectées. Ne pas se préoccuper des objets dont le nom commence par deux underscores, comme `__name__`, il s'agit de noms réservés.
- `dir(module)` donne la liste des fonctions de ce module,
- `help(fonction)` donne l'aide d'une fonction, on peut aussi utiliser `help(type)` qui donne les méthodes d'un type.
- la fonction `locals` permet de lister (sous la forme d'un dictionnaire) les variables et leur valeurs. On peut l'utiliser sous la forme : `print(locals())`

■ **Exemple III.3** Voici un exemple complet :

```
>>> dir() # seul sont présents les noms réservés
['__builtins__', '__doc__', '__loader__', '__name__', '
    __package__', '__spec__']

>>> import math # chargement de math

>>> a=2 # affectation de a

>>> dir() # a et math apparaisse
['__builtins__', '__doc__', '__loader__', '__name__', '
    __package__', '__spec__', 'a', 'math']
>>> dir(math) # listes des fonctions du module math
['__doc__', '__loader__', '__name__', '__package__', '
    __spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', '
    atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', '
    degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', '
    factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', '
    hypot', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma',
```

```
    'log', 'log10', 'log1p', 'log2', 'modf', 'pi', 'pow', '
   radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']

>>> help(math) # aide complète sur un module

>>> help(math.sin) # aide sur une fonction
Help on built-in function sin in module math:

sin(...)
    sin(x)

    Return the sine of x (measured in radians).

>>> print(locals()) # on voit a et sa valeur
{'__name__': '__main__', 'a': 2, '__package__': None, '
   __loader__': <class '_frozen_importlib.BuiltinImporter'
   >, '__builtins__': <module 'builtins' (built-in)>, 'math
   ': <module 'math' (built-in)>, '__spec__': None, '
   __doc__': None}
>>> help(list) # aide sur un type
```

■

III.1 Variables en Python

On a déjà précisé ce que Python appelle variable. Il faut bien avoir en tête la différence entre les objets mutables et les objets immuables.

Une variable est **identifiée** par son nom :

- chiffre et lettre uniquement : pas d'accent, pas d'espace, pas de point, pas de caractères spéciaux. Quelques noms sont interdits.
- Le nom peut contenir des chiffres mais doit commencer par une lettre.
- Les majuscules sont importantes.
- Les constantes nommées sont à mettre en haut et en majuscule.

Les types usuels sont : `int`, `float`, `bool`, `str`, `list`, `tuple` et `array`. Le nom du type est toujours une fonction qui convertit vers le type.

Python fait en particulier la différence entre les entiers et les flottants, même si la conversion est souvent automatique. Avec les entiers le calcul est toujours exact, avec les flottants le calcul est toujours faux.

Il y a un type complexe (appelé `complex`). Un nombre complexe s'écrit sous forme $a+bJ$ ou $a+bj$ avec a et b flottants ou entiers. Le **module** `cmath` contient les fonctions mathématiques utiles pour les complexes.

On peut utiliser l'affectation parallèle :

```
a, b, c = 2, 5, 7 # crée 3 variable en une instruction.
a,b = b,a # échange le contenu de a et b
```

On peut aussi utiliser l'affectation simultanée : `x=y=5` , ce qui permet d'initialiser plusieurs variables.

III.2 Les structures de contrôles en Python

⋆ Banchement conditionnel

Le branchement conditionnel s'utilise avec `if` suivi d'un booléen.

Pour tester si un booléen `B` est vrai, on fait `if B` et non `if B==True`. On peut combiner les tests de manière naturelle en écrivant par exemple `if a<b<=c`.

L'évaluation paresseuse est à connaître : lorsque l'on combine deux booléens : `P and Q` , **l'ordre a de l'importance**, si `P` est `False`, alors Q n'est pas évalué. Cela sert dans le cas où le booléen Q n'est pas défini si P est `False`.

■ **Exemple III.4** On cherche à modéliser un feu dans une forêt. La forêt est modélisée par un tableau `F` à $n \times m$ case. La case (i, j) a pour coordonnée `F[i,j]` pour $(i, j) \in [\![0, n-1]\!] \times [\![0, m-1]\!]$.

Le principe c'est que la case (i, j) est mise à la valeur `FEU` si l'une des 4 cases voisines par les côtés contient la valeur `FEU`. Avec bien sûr des cas particuliers pour les cases du bords. Pour chaque test, on vérifie l'existence de la case avant de regarder son contenu (sinon on a un `Invalid Index`).

On écrit :

```
if (  (i>0 and F[i-1,j] == FEU) # NORD
   or (i<(n-1) and F[i+1,j] == FEU) # SUD
   or (j>0 and F[i,j-1] == FEU) # EST
   or (j<m-1 and F[i,j+1] == FEU) ) # OUEST
F[i,j] = FEU
```

L'ordre des tests a ici beaucoup d'importance. ■

⋆ Utilisation des itérateurs

Un **conteneur** est un objet qui contient plusieurs objets. Un conteneur est **itérable** si il est possible de parcourir les éléments avec une boucle `for`.

Un itérateur est donc un objet où il y a un premier élément, puis un suivant, etc.

```
for elm in iterable :
```

La variable `elm` prends alors successivement les références des valeurs de `iterable`. Il n'y a pas besoin d'initialiser la variable `elm`.

On peut ainsi faire : `for x in liste`, car une liste est itérable. La variable `x` est affectée à la première valeur de L, puis la deuxième, etc. mais aussi : `for lettre in msg` La variable `lettre` prends alors la valeur des lettres de `msg`.

La fonction `range` est une fonction qui crée des itérables sur les entiers :

- `range(stop)` crée l'itérateur 0, 1, jusqu'à `stop -1`,
- `range(start, stop)` crée l'itérateur `start`, `start+1` jusqu'à `stop -1`,
- `range(start, stop, step)` crée l'itérateur `start`, `start+step` jusqu'à `stop` exclu.

> En pratique : il existe donc en fait deux boucles `for` (deux moyens de parcourir une liste) :
>
> - **sur les éléments :** `for x in L`. On utilise l'élément `x`.
> - **sur les positions :** `for i in range(n)` où `n` est la longueur de la liste. L'élément lu est alors `L[i]`.
>
> La première méthode est plus compacte et plus lisible, mais ne permet pas de se

repérer dans la liste. Lorsque l'on veut parcourir les éléments dans leur généralité, on utilise la boucle sur les éléments, lorsque l'on doit repérer où les éléments sont situés, on utilise la boucle sur les positions.

En fait on peut faire les deux avec la fonction `enumerate`, qui prend en entrée une liste `L` et sort la liste $(i, L[i])$ pour $i = 0 \ldots \texttt{len}(\texttt{L}) - 1$.

Lorsqu'on utilise les boucles `for` il y a deux chose à avoir en tête :

- lorsque l'interpréteur lit `for i in range(n):` il construit à l'avance l'itérateur que va parcourir i. Ainsi, modifier n dans le bloc répété, ne modifie donc pas la boucle.
- lorsque l'interpréteur lit `for i in range(n):` la variable i est créée et initialisée à 0. Inutile de l'initialiser À la fin de la boucle, la variable `i` existe encore et sa valeur est $n-1$. Elle n'est pas détruite mais ne devrait pas être utilisée.

■ **Exemple III.5** Les instructions :

```
n=3
for i in range(n) :
     print( "i="+str(i)+" n="+str(n) )
     n += 2
print("a la fin, n="+str(n)+" et i="+str(i))
```

écrivent :

```
i=0 n=3,
i=1 n=5,
i=2 n=7,
a la fin, n=7 et i=2
```

■

L'instruction **`break`** permet de sortir d'une boucle `for`, ou d'une boucle `while` immédiatement mais pas d'un `if`. L'instruction **continue** permet de revenir en début de boucle sans finir d'exécuter le bloc d'instructions. Cette fonctionnalité ne sont à n'utiliser que si vous êtes à l'aise avec les techniques classiques.

✯ Boucle conditionnelle `while`

La boucle `while` est à utiliser quand on ne sait pas à l'avance combien de fois le bloc doit être répété. La syntaxe est :

```
while test :
  bloc répété
La suite
```

En pratique :

- toujours écrire au brouillon la négation de la condition (« jusqu'à ce que »).
- Bien garder en tête que **lorsque l'on sort de la boucle, le test est faux**.
- Vérifier que l'on n'a pas une boucle infinie : une instruction de la boucle doit modifier le test.
- le `while P and Q` est souvent suivi d'un `if P:` on veut savoir si c'est P ou

Q qui est faux.

■ **Exemple III.6** Poser une question et la reposer la question jusqu'à avoir l'une des réponses correctes

```
rep = "" # force à rentrer dans la boucle
while  rep != "o" and rep != "n" :
  rep = input("o ou n?") # modification du test
if rep == "o" :
  print("oui")
else : # pas besoin de elif ici
  print("non")
```

Beaucoup se trompent sur le `and` dans la condition. ■

Attention : le test n'est évalué qu'à la fin du bloc. **Il peut donc passer à `False` à l'intérieur du bloc sans que la boucle s'arrête**. Par exemple, on peut faire :

```
while test :
   test = False
   # suite du bloc d instructions
   # pouvant faire passer test à True.
```

Si le `test` est faux au premier passage, le bloc n'est jamais exécuté. On « **force** » parfois à rentrer dans la boucle, pour exécuter le bloc jusqu'à ce que le test soit faux.

■ **Exemple III.7 Recherche d'un nombre par essais successifs**

Le programme suivant est faux :

```
x = randint(1,15)
y=0 # force à rentrer dans la boucle
while x != y:
    y = int(input("donnez un nombre"))
    if x<y :
        print("trop grand")
    else :
        print("trop petit")
```

En effet, il affiche « trop petit » (une fois) même lorsque l'on donne la bonne réponse !

Le test `x<y` n'est pas évalué en permanence, uniquement en début de boucle.

Il faut faire par exemple :

```
x = randint(1,15); y=0
while x != y:
    y = int(input("donnez un nombre"))
    if x<y :
        print("trop grand")
    elif x> y :
        print("trop petit")
```

■

On considère une boucle `for` que l'on veut transformer en boucle `while` :

```
for i in range(n):
```

```
  instructions
```

Il faut faire :

```
i=0 # on initialise la variable i
while i<n:
  instructions
  i += 1 # on incrémente la variable i
```

Les deux boucles sont indentiques, sauf si on ajoute un test dans le `while`, pour sortir avant la fin de la boucle : `while i<n and test:`

A contrario, initialiser ou incrémenter la variable dans une boucle `for` est une erreur de programmation.

On a ainsi souvent la structure `while P and Q` ou P et Q sont deux expressions booléennes (des propositions) qui est souvent suivi d'un `if P` pour savoir si c'est la condition P ou la condition Q qui a fait sortir de la liste.

III.3 Conteneurs : str, list et array

✯ Chaîne de caractères

C'est le type pour stocker des mots.

- Ce type est indexable : on peut faire `mot[k]`,
- Ce type est itérable : on peut faire `for lettre in mot`,
- Ce type est immuable : toute modification d'une chaîne de caractères entraîne la création d'un nouvel objet.

■ **Exemple III.8** Pour afficher la chaîne `msg` en soulignant en dessous :

```
chaine = msg + "\n" + "-"*len(msg)
print(chaine)
```

■

✯ Listes

Le type `list` sert à stocker des objets de type quelconque de taille quelconque.

- Ce type est indexable : on peut faire `liste[k]`,
- Ce type est itérable : on peut faire `for x in liste`,
- Ce type est mutable.

Une liste peut contenir tout type de données dont une liste.

Voici une liste des opérations sur les listes :

- Accéder à la valeur d'un indice : en utilisant `L[k]` pour k entre 0 et `len(L)-1`
  On a aussi `L[-1]` qui est le dernier élément, `L[-2]` l'avant dernier, etc.
- Ajouter un élément à la fin : `L.append(elmt)`.
  Ajouter une liste à la fin : `L.extend(L2)`. On peut aussi faire `L += L2`.
- Insérer un élément à la place i : `L.insert(i, elmt)`
- On peut **supprimer l'élément `k`** : `del(L[k])`. La taille de la liste est alors automatiquement diminuée de 1.

- Dépiler le dernier élément (structure de pile) : `y = L.pop()`. Celui-ci est alors affecté à la variable *y*.
- « déballer » (*unpacking*) les éléments : `a, b, c = L`, *a* est la alors le premier élément de *L*, *b* le deuxième et *c* le troisième.
  On peut aussi utiliser :

```
[a, b, c] = L # avec une liste
(a, b, c) = L # avec un Tuple
```

  Cette propriété sert en particulier pour récupérer la sortie d'une fonction qui renvoie plusieurs valeurs (en fait une liste de valeurs) :

```
# pour récupérer la taille d'une image = deux valeurs:
n, m = tailleImage(img)

# on peut aussi faire:
t = tailleImage(img) # t est un couple d'entiers
n = t[0]; m = t[1] # nbr de  lignes, colonnes
```

  Elle sert aussi lorsqu'on parcours une liste de couple (ou de triplet). Par exemple :

```
for ii,jj in listePixel:
```

  Ce qui est équivalent à :

```
for pixel in listePixel:
    ii=pixel[0]; jj=pixel[1]
```

- On peut tester l'appartenance. Pour un élément *x* et une liste *L*, on dispose de l'opérateur `in` qui renvoie un booléen : `True` si $x \in L$, `False` sinon.
  On peut donc faire : `if 0 in liste:` ou `while 0 in liste:`. La négation est `if 0 not in liste:` ou `while 0 not in liste:`
- On a d'autres méthodes :
  - `L.index(value)` retourne le premier indice *i* tel que `L[i]=value`
  - `L.remove(value)` enlève la première occurrence de la valeur `value`
  - `L.reverse()` renverse la liste.
  - `L.sort()` trie la liste.
  - `nbr = L.count(value)` retourne le nombre d'indice de `L` égaux à `value`
  - `L2 = L.copy()` copie la liste

La plupart du temps, l'indexation négative est source d'erreurs.

Listes en compréhension

La syntaxe générale est :

```
[exp for variable in liste]
[exp for variable in liste if condition]
```

avec :

- `liste` un **itérable**, `variable` parcourt cet itérable.
- `exp` une expression qui peut dépendre de `variable`,
- `condition` une **expression de type booléen** qui peut dépendre de `variable`.

On obtient alors **la liste des expressions** lorsque `variable` décrit `liste` si `condition` est vrai.

■ **Exemple III.9** La liste des carrés de $[\![0,9]\!]$:

```
[i**2 for i in range(10)]
```

La liste des carrés des nombres pairs de $[\![0,9]\!]$:

```
[i**2 for i in range(10) if i%2==0]
```

```
[ [i, i+1] for i in range(6) ]
```

crée la liste

```
[[0, 1], [1, 2], [2, 3], [3, 4], [4, 5], [5, 6]]
```

■

On peut mettre plusieurs boucles for avec la syntaxe :

```
[exp for v1 in l1 for v2 in l2 if condition]
```

■ **Exemple III.10** Les instructions :

```
[ [i,j] for i in range(1,7) for j in range(1,7) if i<j]
```

créent la liste des couples (i, j) avec $(i, j) \in [\![1,6]\!]$ et $i < j$.

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

■

■ **Exemple III.11** L'instruction :

```
[ l*j for l in "abc" for j in range(1,4)]
```

crée la liste :

```
['a', 'aa', 'aaa', 'b', 'bb', 'bbb', 'c', 'cc', 'ccc']
```

■

L'expression peut aussi être elle même une liste en compréhension, pour faire des listes de listes

■ **Exemple III.12** L'instruction

```
[ [ 0 for i in range(n)] for j in range(m)]
```

crée une liste de liste contenant m listes de taille n, dont tous les éléments sont nuls. ■

■ **Exemple III.13** L'instruction

```
[ [ i+1 for i in range(j)] for j in range(1,7)]
```

Crée la liste de liste :

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

■

En pratique :

- Les listes en compréhension sont **un moyen rapide et pratique de créer des listes**.
- La syntaxe est assez naturelle et corresponds à la description des ensembles en mathématiques sous la forme d'une image directe :

$$f(E) = \left\{ f(x) \middle| x \in E \right\}.$$

- On peut toujours s'en passer : la liste en compréhension

```
L = [expression for var in liste if condition]
```

est équivalente aux instructions :

```
L = []
for var in liste :
  if condition :
    L.append(expression)
```

■ **Exemple III.14** On veut extraire d'une liste `L` de `float` la sous-liste constituée des termes positifs strictement. Par une liste en compréhension :

```
LL = [x for x in L if x>0]
```

Par la concaténation :

```
LL = []
for x in L :
  if x>0 :
    LL.append(x)
```

■

Extraction de sous-listes et de sous-matrices (Slicing) On peut extraire rapidement une partie d'un objet indexable.

La syntaxe est naturelle :

`extrait=liste[start:stop]`, de `start` à `stop` exclu.

On peut aussi faire :

```
liste[:stop] # start = 0 par défaut
liste[start:] # stop = len(liste) par défaut
liste[:] # tous pour copie
liste[start:stop:step]
```

■ **Exemple III.15** On dispose de deux listes L_1 et L_2.

On veut mettre dans une liste le premier élément de L_1 devant puis tous les éléments de L_2 :

```
L = [ L1[0] ] + L2[:]
```

Les premiers éléments de L_1 suivi des derniers de L_2, puis les derniers de L_1 (dans l'ordre inverse) et enfin les premiers de L_2 (dans l'ordre inverse)

```
L = L1[:4] + L2[4:] + L1[:4:-1] + L2[3::-1]
```

■

On utilise souvent le slicing en utilisant un index négatif.

```
 +---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

■ **Exemple III.16** On veut savoir si le nom du fichier finit par `.pdf`, on écrit alors :

```
nomFichier[-4:] ==".pdf"
```

■

✯ Array

Pour stocker un tableau d'éléments de même nature, et de taille fixée on utilise un array. Celui-ci peut être en une dimension, ie un vecteur (1d-array), en deux dimension ie un tableau (2d-array) et même en trois dimension, ie une hypermatrice (3d-array).

Pour créer un tel tableau, il faut utiliser des bibliothèques. On utilise les fonctions `ones`, `zeros` de la bibliothèque `pylab` (ou de la bibliothèque `numpy`).

```
 # dans la partie bibliothèque et fonctions importées:
from pylab import ones, zeros

tab = ones( (n,m) )
# crée un tableau de réels de taille (n,m) rempli de 1
tab = zeros( (n,m) )
# crée un tableau de réels de taille (n,m) rempli de 0
vec = zeros( n )
# crée un vecteurs nuls de taille n
vec = zeros( (n,m,3) )
# crée une hypermatrice de taille (n,m,3)
#       = une image couleur

tab = ones( (n,m), int)
# crée un tableau d'entiers

tab = array([ [1,2,3],[4,5,6] ])
#convertit une liste de liste en 2d-array
```

La taille du tableau s'obtient par la fonction `shape` :

```
n, m = shape(tab) # pour les 2d-array
n = shape(vec) # pour les 1d-array
```

On peut accéder aux éléments comme avec les listes :

```
tab[i,j] # pour les 2d-array
vec[i] # pour les 1d-array
```

Le principal avantage des `array` sur les listes est que les opérations sont naturelles :

- une somme de `array`+`array` corresponds à la somme terme à terme,
- une somme de `float`+`array` corresponds à ajouter une valeur à chaque terme
- un produit `float`×`array` correspond à multiplier tous les termes par le scalaire.

On peut dire que les `array` sont les éléments de $\mathbb{R}^n$ (on a les mêmes opérations que pour un espace vectoriel), tandis que les listes sont des n-uplets.

Voici un tableau comparant les listes et les 2d-array.

Tableau	**Liste de listes**
nombre d'éléments fixés	concaténation
éléments de même type	éléments de type quelconque
`*` = multiplier `+` = ajouter à tous les termes `==` = comparaison terme à terme	`*` = répéter `+` = concaténer `==` = comparaison globale
`tab[i,j]`	`L[i][j]`
`n,m=shape(tab)`	`n=len(L);m=len(L[0])`

✯ Utiliser des 2d-array pour manipuler des matrices

On utilise souvent des 2d-array pour les matrices.

Le produit de deux 2d-array est celui du produit tableau (termes à termes) et non le produit matriciel qui s'obtient par la fonction `dot`. Attention aussi : $A+5$ est interprété comme `A+5*ones([n,n])` avec (n,n) la taille de A, et non comme $A+5I_n$.

Pour les algorithmes du type pivot de Gauss, il est très pratique d'utiliser l'extraction d'une ligne / colonne :

```
Li = mat[i,:] # extrait la ligne i
Cj = mat[:,j] # extrait la colonne j
```

On peut ainsi faire les opérations élémentaires :

- Échange de lignes : $l_i \leftrightarrow l_j$

```
mat[i,:], mat[j,:] = mat[j,:], mat[i,:]
```

- Multiplication d'une ligne par un scalaire : $l_i \leftarrow \beta l_i$

```
mat[i,:] = beta*mat[i,:]
```

- Opération élémentaire : $l_i \leftarrow l_i + a l_j$

```
mat[i,:] += a*mat[j,:]
```

✯ Utilisation des 1d-array pour dessiner des fonctions

Pour dessiner des courbes en Python, il faut construire deux listes (ou deux 1d-array ou encore deux itérateurs) : la liste des abscisses $[x_0,\dots,x_{n-1}]$ (souvent des points équirépartis) et la liste des ordonnées $[f(x_1),\dots,f(x_n)]$.

Python dessinera alors les segments de droites qui relient les n points de coordonnées $\Big((x_i,f(x_i)\Big)$. Si il y a suffisamment de points, la courbe ressemblera à une courbe lisse et non à des segments de droites.

Pour créer les listes de points équirépartis, on dispose de deux fonctions à connaître :

- `linspace(xmin,xmax,nbrPoints)` (« *linéairement espacée* ») qui crée un `array` de taille `nbrPoints`, dont le premier élément est `xmin` et le dernier `xmax`.
  On contrôle ainsi le nombre de points de la discrétisation de l'intervalle [`xmin`, `xmax`], le pas est : $\dfrac{\texttt{xmax}-\texttt{xmin}}{\texttt{nbrPoints}-1}$.
- `arange(xmin,xmax,pas)` (« *array range* ») : crée un `array` dont le premier élément est `xmin` et dont chaque élément est distant de `pas`, le dernier étant strictement inférieur à `xmax`.
  Cette fonction est donc semblable à `range` sauf que l'on accepte ici un pas non entier, et que la sortie est un `array`.
  On contrôle ainsi le pas de la discrétisation de l'intervalle [`xmin`, `xmax`[.
  Attention : le dernier point n'est jamais `xmax`.

Pour créer la liste des images, on peut utiliser les opérations sur les `array`. On rappelle que si `x` et `y` sont des `array` et `alpha` est un scalaire, on a :

- `x+y` est obtenu en ajoutant terme à terme les éléments de `x` et `y`,
- `x+alpha` est obtenu en ajoutant `alpha` à tous les termes,
- `alpha*x` est obtenu en mutlipliant tous les termes par `alpha`,
- `x*y` est obtenu en mutlipliant terme à terme les éléments de `x` et `y`,
- `cos(x)` est obtenu en appliquant la fonction cosinus à tous les éléments de `x`. C'est le comportement pour toutes les fonctions mathématiques (fonctions universelles présentes dans la bibliothèque `numpy`). **Attention :** pour appliquer une fonction qui n'est pas dans la bibliothèque, il faut revenir à la boucle `for`.
- `x**n` est obtenu en mettant à la puissance n-ième tous les éléments de x.
- `1/x` est obtenu en prenant l'inverse de tous les éléments de x.

Ces fonctionnalités permettent de calculer la liste des images très facilement à partir de la liste des indices.

R Sauf indication contraire, il est plus simple de travailler avec `linspace`.

■ **Exemple III.17** si on doit dessiner la courbe représentative de la fonction :

$$x\longmapsto \frac{x^3+3x+2}{x^2+1}+x\sin(x)+e^x.$$

sur l'intervalle $[-10,10]$, on écrit simplement :

```
x = linspace(-10, 10, 100)
y =  (x**3+3*x+2) / (x**2+1)  + x*sin(x) + exp(x)
```

```
plot(x,y)
show()
```

■

■ **Exemple III.18** De même, le dessin du cercle peut se faire avec :

```
theta = linspace(0,2*pi,100)
plot(cos(theta), sin(theta))
```

■

Pour dessiner une courbe paramétrée définie par $(x(t),y(t))$ pour $t \in I$, il suffit de discrétiser l'intervalle I en n valeurs $t_1,\dots y_n$, et de créer deux listes (ou deux `1d-array`) x et y de taille n contenant les valeurs de $[x(t_1),\dots x(t_n)]$ et $[y(t_1),\dots,y(t_n)]$.
La courbe est simplement affichée par la commande `plot(x,y)`.

■ **Exemple III.19** Par exemple, pour la courbe

$$\begin{cases} x(t) = & \cos(t)\,(\sin(t)+1) \\ y(t) = & \sin(t)\,(\cos(t)+1) \end{cases} \qquad \text{avec } t \in [0,2\pi]$$

On peut faire

```
# liste des t: équi-répartis entre 0 et 2%pi
t = linspace(0, 2*pi, 300)

# liste des (x,y) correspondants:
x = cos(t) * (sin(t)+1)
y = sin(t) * (cos(t)+1)
plot(x,y)
show()
```

■

■ **Exemple III.20** Un autre exemple pour la courbe $\begin{cases} x(t) = & \cos(t)\cos\left(\frac{t}{2}\right) \\ x(t) = & \cos(t)\sin\left(\frac{t}{2}\right) \end{cases}$ avec $t \in [0,4\pi]$.

```
t = linspace(0,4*pi,300)
x = cos(t) * cos(t/2)
y = cos(t) * sin(t/2)
plot(x,y)
show()
```

■

III.4 Les fonctions

Pour créer une fonction la syntaxe est :

- `def nomFonction(x1,x2)` : les entrées sont les variables x1,x2
- suivi **obligatoirement** par une description des entrées et sorties en utilisant les « doc string » `"""`
  Pour chaque entrée / sortie, on veut le type (informatique) et l'interprétation.

- le bloc d'instructions de la fonction est délimité par : et l'indentation, et permet de **calculer les sorties**, les variables `x1,x2` étant supposées connues.
- `return y1,y2,y3` sort de la fonction et **retourne les valeurs des expressions** `y1,y2,y3`.
- On utilise la fonction sous la forme : `a,b,c = nomFonction(u,v)` avec `u,v` des expressions, et `a,b,c` des variables.

Une fonction est faite pour être appelée différentes entrées et sa sortie est faite pour être utilisée de différentes manières. Si l'entrée de la fonction est x et la sortie y, on peut tout à fait l'utiliser sur l'entrée t et appeler la sortie u.

En particulier, une fonction ne contient normalement pas d'instructions `print` (puisqu'elle est faite pour être appelée plusieurs fois sur plein d'entrées différentes). L'affichage des résultats de la fonction se fait dans l'espace appelant. Il y a deux exceptions :

- lorsqu'on cherche les erreurs dans le code, on ajoute bien sûr des instructions `print` temporaires,
- certaines fonctions sont faites pour afficher les résultats (graphiques) ou impression à l'écran. Dans ce cas, elles ne renvoient rien.

De même, d'une manière générale, une fonction ne contient pas de `input`. Si une fonction a besoin d'une valeur pour s'exécuter, celle-ci doit être en entrée. D'une manière générale, il est souvent plus simple d'écrire les valeurs dans le script que de faire un input.

✯ Instruction `return`

L'instruction `return` sert à sortir d'une fonction en renvoyant une ou plusieurs valeurs. En fait, il n'y a qu'une valeur de sortie mais qui peut être un tuple contenant plusieurs valeurs. Une fonction sans `return` renvoie en fait un type particulier : `None`.

Une fonction peut contenir plusieurs `return`. Elle se termine au premier `return` évalué.

■ **Exemple III.21** Dans le code suivant, on n'a pas besoin de `else`

```
def signe(x):
  if x>=0:
    return 1
  return 0
```

■

La principale utilisation de cette fonctionnalité est que l'on évite les accumulations de `if`. On gagne ainsi en lisibilité. On peut dire que l'on traite le cas particulier, puis on revient au cas général. Ce qui est particulièrement le cas pour les algorithmes récursifs. Parfois cela permet de gagner du temps de processeur : on sort de la fonction dès que l'on a l'information importante.

✯ Variable fonction

Une fonction est une variable comme les autres. Si on définit la fonction `f` alors on ne peut pas utiliser une autre variable nommée `f` ni dans la fonction ni ailleurs. Une fonction peut aussi être une entrée d'une autre fonction.

■ **Exemple III.22** On veut résoudre $f(x) = 0$. On écrit une fonction `dichotomie`, dont la définition est : `def dichotomie(f,a,b,eps)` : l'entrée de la fonction `dichotomie` est une fonction appelée `f` ■

✯ Entrées d'une fonction

On met en entrée les variables dont a besoin la fonction pour produire un résultat et uniquement celle-là : par exemple si une fonction agit sur une liste, ne pas mettre la taille de la liste en entrée. Elle est calculé dans la fonction avec `n = len(L)`.

En pratique, la structure conseillée pour un script est :
- Les `import` depuis les autres modules.
- Les constantes nommées, ie les variables dont la valeur n'évolue pas au cours de l'exécution du script et qui sont utilisées dans les fonctions sans être en entrée.
- Les fonctions (suivies systématiquement d'un commentaire en doc string)
- Des instructions qui permettent de tester rapidement le travail effectué.

Il est conseillé de faire des scripts « autonomes » : lorsqu'on exécute le script, les instructions montrent proprement que les fonctions sont correctes.

Pensez à soigner vos affichages : un script qui affiche plein de lignes illisibles ne sert à rien. Chaque affichage doit être bien lu.

■ **Exemple III.23** Voici un exemple de script

```
import numpy  as np

GRAVITE= 9.81

def simule( alpha, vit)
  """
  entrée: alpha = float
                = angle de lancer avec l'axe horizontal
          vit = float
              = vitesse initiale de l'objet
  sortie: x_impact = float
          = abscisse de l'impact

  Cette fonction simule le déplacement d'un objet
  lancé depuis l'origine avec une certaine vitesse
  initiale et calcule l'abscisse de l'impact
  """
  #####

print("--- début du programme ---")

# test sur tous les angles de pi/8 à 3*pi/4
# avec une vitesse de 1.5
for alp in linspace(np.pi/8, 3*np.pi/4):
  imp = simule(alp, 1.5)
```

```
26    print("avec alpha =", alp, "on a un impact en :", imp)

28
print("--- fin du programme ---")
```

■

IV Algorithmes à connaître

■ **Exemple IV.1** Utilisation de la concaténation : on part du vide et on ajoute dessus :

```
nbr = int( input("donnez un nbr"))
L = []
msg = ""
somme = 0
while nbr != 0 :
  L.append(nbr)
  msg +=  str(nbr)+" "
  somme += nbr
  nbr = int( input("donnez un nbr (0 pour stop)"))
print("vous avez entré:" + msg)
print("sous forme de liste:", L)
print("le total est :" + str(somme))
```

On peut voir que la variable `msg` contient un espace en trop à la fin. ■

■ **Exemple IV.2** Calcul des termes de la suite de Fibonnacci

$$F_0 = 0, \quad F_1 = 1, \quad \text{et} \quad \forall n \in \mathbb{N},\ F_{n+2} = F_{n+1} + F_n.$$

En calculant la liste des termes de la suite :

```
F = [0]*n # initialisation d'une liste F
F[1] = 1
for i in range(2,n) :
    F[i] = F[i-1] + F[i-2]
```

En ne gardant en mémoire que les deux derniers :

```
  Fnm1 = 0 # valeur de Fn-1
  Fn = 1 # valeur de Fn
  for i in range(1,n) :
    # Fn devient Fn+1
    # et Fn-1 devient Fn
    Fn , Fnm1 = Fn + Fnm1 , Fn
```

■

■ **Exemple IV.3** Recherche d'un maximum : boucle sur les indices.

```
maxi = L[0]
indMaxi = 0
for i in range(len(L)) :
    if L[i] > maxi :
        indMaxi = i
        maxi = L[i]
```

Recherche d'un maximum : boucle sur les éléments

```
maxi = L[0]
for x in L :
    if x > maxi :
        maxi = x
```

L'avantage de la première méthode est qu'elle permet d'obtenir l'indice de l'élément maximal.

Avec `enumerate` on combine les deux :

```
maxi = L[0]
imax = 0
for i,x in enumerate(L):
  if x>maxi:
    maxi = x; imax = i
```

■

■ **Exemple IV.4** On dispose d'une liste `L` de taille n, on veut construire une liste `M` de taille n telle que :

$$\forall i \in [\![1, n-2]\!], \quad \texttt{M[i]} = \frac{\texttt{L[i-1]} + 2\texttt{L[i]} + \texttt{L[i+1]}}{4}$$
$$\texttt{M[0]} = \texttt{L[0]} \qquad \text{et} \qquad \texttt{M[n-1]} = \texttt{L[n-1]}$$

Il faut alors boucler sur **la position**

```
M = [0]*n
M[0] = L[0]
M[n-1] = L[n-1]
for i in range(n-1) :
    M[i] = (L[i-1] + 2*L[i] + L[i+1]) /4
```

■

■ **Exemple IV.5 Recherche d'un élément nul avec un for**

- On initialise une variable « drapeau » `contient0` à `False`
- On lit tous les éléments (avec `for`), si on trouve un élément nul, on fait passer `contient0` à `True`
- Attention : on ne remet jamais `contient0` à `False`.

Voici la version « boucle sur les positions »

```
contient0 = False
for i in range(n) :
  if L[i] == 0 :
    contient0 = True
if contient0 :
  print("un élément nul au moins")
else :
  print("pas d'élément nul")
```

et la version « boucle sur les éléments » :

```
contient0 = False
for x in L :
  if x==0 :
    contient0 = True
```

Enfin, on peut utiliser une fonction avec plusieurs `return` :

```
def contient0(L) :
  """
  entrée: L = list
  sortie: True si  0 in L, False sinon
  """
  for x in L :
    if x== 0:
      return True
  return False
```

■

■ **Exemple IV.6 Recherche d'un élément nul avec un while**

- On pose la tête de lecture sur le premier élément,
- tant que ce que l'on lit existe et ne convient pas, on déplace la tête de lecture.

```
contient0 = False
i = 0
while i<n and not(contient0) :
    if L[i] == 0 :
        contient0 = True
    i += 1
```

■

■ **Exemple IV.7** On dispose d'une liste `L`, on souhaite savoir si la liste est croissante. On va utiliser **une variable booléenne** (drapeau) qui va passer à faux si deux éléments consécutifs sont dans le mauvais ordre.

```
estCroissante = True
for i in range(n-1) :
    if L[i]>L[i+1] :
        estCroissante = False
```

La même chose avec un while : on pose la tête de lecture sur les deux premiers éléments, tant que ce que lit la tête de lecture est dans le bon sens, on la déplace :

```
estCroissante = True
i = 0
while i<n-1 and estCroissante :
    if L[i]>L[i+1] :
        estCroissante = False
    i += 1
```

■

■ **Exemple IV.8 Égalité de listes**

On considère deux listes `l1` et `l2` de même longueur, on veut savoir si elles sont égales.

Avec une boucle `for` et un « drapeau » qui passe à `False` :

```
egalite = True
for i in range(len(l1)):
  if l1[i]!=l2[i]:
    egalite = False
```

Avec une boucle `while` :

```
egalite = True
i = 0
while i<len(l1) and egalite:
  if l1[i]!=l2[i]:
    egalite = False
  i += 1
```

La boucle `while` est ici plus rapide.

Avec une fonction, utilisant deux `return` :

```
def listeEgale(l1,l2):
  if len(l1)) != len(l2):
    return False
  for i in range(len(L1)):
    if l1[i] != l2[i] :
      return False
  return True
```

■

■ **Exemple IV.9 Tirage de 8 cartes dans un jeu**

Le principe est de créer une liste avec les carte du jeu (classées), puis de retirer de cette liste pour insérer dans la main :

```
jeu = creeJeu() # initialisation dans un ordre quelconque
main = []
for i in range(8):
    k = randint(len(jeu)) # un indice au hasard
    main.append(jeu[k])
    del(jeu[k]) # ajout et effacement.
```

On pouvait aussi utiliser la fonction pop :

```
jeu = creeJeu() # initialisation dans un ordre quelconque
main = []
for i in range(8):
    k = randint(len(jeu)) # un indice au hasard
    carte =jeu.pop(k)
    main.append(carte)
```

■

■ **Exemple IV.10 Test d'appartenance :**

```
listeUtilisateurs = ["toto", "dupont",
rep = "" # force à rentrer dans la boucle
while rep not in listeUtilisateurs :
    rep = input("quel est ton nom?")
```

Dans un autre genre :

```
rep = input("continuer, oui ou non?")
if rep in ["o", "oui", "Oui", "OUI" ] :
   print("on continue")
elif rep in ["n", "non", "Non", "NON" ] :
   print("on stoppe")
else :
   print("bah alors ?")
```

■

V Ingénieurie numérique

V.1 Calcul de moyenne et de variance

On dispose d'une liste `l` que l'on interprète comme une série statistique. On souhaite calculer la moyenne $\bar{l}$ et la variance $\sigma^2(l)$.

$$\begin{aligned}\bar{l} &= \frac{1}{n}\sum_{k=0}^{n-1} l_i \\ \sigma^2(l) &= \frac{1}{n}\sum_{k=0}^{n-1} \left(l_i - \bar{l}\right)^2 \\ &= \overline{l^2} - \left(\bar{l}\right)^2\end{aligned}$$

si on a la listes des réalisations de la variable X (ie la liste des résultats des expériences) :

```
def moyVar(l):
  """
  entrée:
  l = list
    = listes des réalisations de la variable aléatoire X
      (série statistique)
  sortie: = couple de float
          = moyenne, variance
  """
  somme, sommeCarre = 0, 0
  n = len(l)
  for elmt in l :
    somme += elmt
    sommeCarre += elm**2
  return somme/n, sommeCarre/n - (somme/n)**2
```

À noter que l'on a parfois non pas la liste des réalisations mais la liste des effectifs. Par exemple, si on suppose que $X(\Omega) = [\![0, n-1]\!]$, et que l'on a la liste L des effectifs : $L[k]$ est le nombre d'expériences où la variable X a été égale à k.

On peut aussi dire que l'on a accès à la liste des fréquences (en divisant chaque valeur de L par le nombre d'expériences).

R La lite des fréquences est la probabilité empirique Si on fait N expériences et que l'on a eu $L[k]$ fois la résultat k, alors la fréquence : $f_k = \frac{L[k]}{N}$ est la valeur mesurée de $p(X = k)$. Souvent on représente cela avec un histogramme.

Cela donne :

```
def moyVar(L, nbrExp):
  """
  entrée:
  L = list
    = listes des effectifs de la série statistique:
      l[k] = nbr d'expé où la variable X a été égale à k
  nbrExp = int
         = nbr d'expérience réalisée
         NB: somme_k( L[k] ) = nbrExp
  sortie: = couple de float
          = moyenne, variance
  """
  n = len(L)

  freq = [L[k] / nbrExp for k  in range(n)]
  # liste des fréquences

  moy, moyCarre = 0, 0
  for k in range(n)
    moy += k*freq[k]
    moyCarre += (k**2)*freq[k]
  return moy, moyCarre - moy**2
```

! À adapter ! En effet, parfois $L[k]$ contient le nombre d'expériences où le résultat est $k-1$.

V.2 Calcul de π par la méthode de Monte-Carlo

On génère n couples (x,y) uniformément dans $[0,1]^2$. On regarde combien de couples vérifient $x^2+y^2 \leqslant 1$.

```
from numpy.random import rand
from math import pi
nbrTest = 10000000
compt = 0
for i in range(nbrTest) :
    x,y = rand(), rand()
    if x**2+y**2 <= 1 :
        compt += 1
print(compt*4/nbrTest, pi)
```

On voit que l'on a convergence :

```
3.1424324 3.141592653589793
```

V.3 Calcul approché d'intégrale

Soit $f : [a,b] \to \mathbb{R}$ une fonction.
On souhaite donner une valeur approchée à $\int_a^b f(t)dt$.

On peut utiliser la **méthode des rectangles :**

- On discrétise le segment $[a,b]$ sous la forme de N points équirépartis :

$$\forall i \in [\![0, N-1]\!],\ t_i = a + i\frac{(b-a)}{N-1}$$

  le pas de la discrétisation est $\Delta = \frac{(b-a)}{N-1}$
- On pose :

$$\int_a^b f(t)dt \approx \Delta \sum_{i=0}^{N-2} f(t_i)$$

Cela peut s'interpréter comme une moyenne.

```
# construction de la discrétisation dans un 1darray
t = linspace(a,b,N)
# pas de la discrétisation
delta = (b-a) / (N-1)
# calcul de la somme:
S = 0
for i in range(N-1):
  S += f(t[i])
S =  S * delta
```

On peut adapter facilement cette technique au calcul approché par des trapèzes.

$$\int_a^b f(t)dt = \Delta\left(\frac{f(a)+f(b)}{2} + \sum_{i=1}^{N-2} f(t_i)\right)$$

V.4 Résolution de $f(x) = 0$

⋆ Algorithme de dichotomie

La méthode de dichotomie est la méthode la plus simple pour résoudre une équation qui s'écrit : $f(x) = 0$.

On dispose d'une fonction $f : [a,b] \to \mathbb{R}$ continue telle que $f(a)f(b) < 0$ (ie de signe contraire). On calcule alors le **point milieu** : $c = \frac{a+b}{2}$. On utilise alors le principe suivant :

- si $f(c)f(a) > 0$, on remplace a par c, et on recommence,
- si $f(c)f(b) > 0$, on remplace b par c, et on recommence.

À chaque itération la taille de l'intervalle est diminuée par 2. On choisit de s'arrêter lorsque $b-a$ est inférieure à une précision donnée.

Voici un exemple d'implémentaton en version itérative

```
def dichotomie(f,a,b,eps)
  """
```

```
  entrée: f fonction
          a,b réel avec a<b et f(a)f(b)<0
          eps>0 précision
  sortie: c valeur approchée d'une solution
                de f(c)=0 à 2 epsilon près
  """
  while (b-a>eps) :
    c=(a+b)/2      # c milieu de l'intervalle
    if f(c) == 0 : #cas facultatif: f(c)=0
      return c
    if f(a)*f(c) > 0 :
      a=c
    else
      b=c
  return c
```

R Cet algorithme peut être utilisé (en l'adaptant) pour l'étude numérique de suites implicites.

On peut aussi écrire des versions qui évitent les appels à la fonctions f en gardant en mémoire les valeurs de $f(a)$ et $f(b)$.

■ **Exemple V.1** On veut calculer les termes de la suites définies par :

$$\forall n \in \mathbb{N},\ u_n^n + u_n^2 + 2u_n - 1 = 0 \qquad u_n \in [0,1].$$

```
def resoudEn(n, eps):
    """
    entrée: n = entier
            eps = une précision
    sortie: un = valeur approchée de la solution de
               En à eps près.
    on calcule un par dichotomie
    """

    a = 0
    fa = a**n + a**2 + 2*a -1 # valeur de f(a)
    b = 1
    fb = b**n + b**2 + 2*b -1 # valeur de f(b)

    while b-a>eps :
      c = (a+b) / 2
      fc = c**n + c**2 + 2*c -1
      if fa*fc >0 :
        a = c
        fa = fc
      else :
        b = c
        fb = fc
    return c
```

■

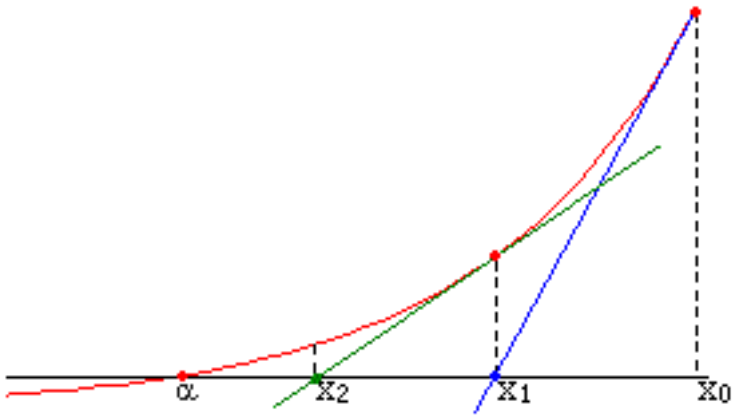
✯ Algorithme de Newton

Considérons une fonction f définie, et dérivable sur un intervalle I. On cherche une solution de $f(x) = 0$.

On suppose que l'on dispose d'une valeur x_0 « proche » de la solution.

Une technique consiste à construire la suite (x_n) en partant de x_0 avec la méthode suivante :

- On considère la tangente T_n en x_n,
- on choisit pour valeur de x_{n+1} la l'intersection de T_n et de l'axe horizontal.
- On choisir de s'arrêter lorsque la suite n'évolue plus.



Plus précisément, l'équation de T_n est :

$$T_n : \qquad y = f(x_n) + f'(x_n)(x - x_n).$$

Ainsi, x_{n+1} est la solution de l'équation : $f(x_n) + f'(x_n)(x - x_n) = 0$, ie :

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

Voici un exemple de code :

```
def newton(f,df, x0, eps) :
    """
    entrée: f = fonction
            df = fonction = dérivée de f
            x0 = réel = point de départ
            eps = réel = précision
    sortie  x = réel
              = valeur approchée de la solution de f(x)=0
    on choisit ici de s'arrêter si on |x_{n+1} - x_n| < eps
    """
    xp = x0 # xp = point précédent
    x = xp - f(xp) / df(xp)
    while abs(xp -x) > eps :
        xp = x
        x = x - f(x) / df(x)
    return(x)
```

V.5 Recherche

On a déjà vu l'algorithme de recherche simple dans une liste (avec plusieurs versions : `for`, `while` ou plusieurs `return`). On a aussi vu les algorithmes recherche de maximum / minimum.

✯ Recherche par dichotomie dans une liste triée

On considère une liste `L` triée par ordre croissant et un élément x, et n cherche à savoir si x est dans `L`. On fait donc une recherche dichotomique :

```
def rechercheDico(x, l) :
  """
  entrée: l = list
          = liste triée par ordre croissant
          x = un élmt
          = on veut savoir si x est dans l
  sortie: True / False selon les cas
  """
  deb, fin = 0, len(l)-1
  while fin -deb> 1 :
    mid = (fin-deb) // 2
    if l[mid] == x :
      return True
    elif l[mid] < x :
      deb = mid
    else:
      fin = mid
  return False
```

✯ Recherche d'un mot dans une chaîne de caractères

On considère un mot `m` de longueur l à chercher dans un texte `t`. On compare ainsi le mot `m` à la portion de texte `t[i:i+l]` pour différentes valeurs de i.

```
def rechercheMon(t, m):
  """
  entrée: t = str
          = texte assez long
          m = str = mot
          on cherche à savoir si m est dans t
  sortie: res = list de int
              = liste des indices i où m est égal à t[i:i+l]
  """
  n, l = len(t), len(l)
  res = []
  for i in range(n-l+1):
    if m == t[i: i+l] :
      res.append(i)
  return res
```

V.6 Équation linéaire

⋆ Méthode de remontée

On veut résoudre le système $AX = b$ pour une matrice A triangulaire supérieure et inversible. On commence par calculer le dernier élément :

$$x_n = \frac{b_n}{a_{n,n}}$$

Puis on remonte :

$$x_i = \frac{1}{a_{i,i}} \left(b_i - \sum_{k=i+1}^{n} a_{i,k} b_k \right)$$

R Attention au décalage d'indice.

```
def resoudRemontee(A,b)
    """
    entrée: A = array([n,n])
         = matrice triangulaire supérieure inversible
            b = array(n) = vecteur
    sortie: X = array(n) = vecteur solution de AX=b
    """

    n,m = shape(A)
    k = len(B)
    if n!=m or k !=n :
        print("pb de taille!")
        return()

    X = zeros(n)

    x[n-1] = b[n-1] / A[n-1, n-1] # dernier élément

    for i in range(n-2,-1,-1):
        # on calcule la somme
        somme=0;
        for k in range(i+1,n)
            # nb: x[k] est déjà calculé
            somme +=   A[i,k]*x[k]
        x[i]= ( b[i] -somme ) / A[i,i]
```

R L'initialisation du dernier élément à part n'est pas utile : si $i = n-1$, la boucle `for k in range(i+1,n)` ne contient aucun terme, l'exécution ne rentre alors pas dans cette boucle.

⋆ Algorithme de Gauss

L'algorithme de réduction de Gauss consiste à réduire le système $AX = B$ en $UX = C$ avec U échelonnée. Conformément au programme, on se contente d'un cas simple : la matrice A est carrée et inversible. Le système est alors de Cramer.

```
def gauss(A, b) :
  """
  entree: A = array
            = matrice carrée inversible de taille nxn
          b = array
            = vecteur second membre.
  sortie: A = array
            = matrice triangulaire supérieure
          b = array
            = vecteur second membre.
  le système AX=b initial est équivalent au système final
  (qui se résout par remontée)
  """
  n, m = shape(A)
  for j in range(n):
    #on traite la colonne j

    #étape 1: on échange les lignes
    if A[j,j] != 0:
      # on cherche le premier terme non nul
      # dans la colonne j en dessous de ajj:
      # comme A est inversible, on est sûr de trouver.
      k =  chNonNul(A,j)
      # on échange lk et lj
      A[k,:], A[j,:] = A[j,:], A[k,:]
      b[k], b[j] = b[j], b[k]

    # arrivé ici, A[j,j] != 0

    #étape 2: on modifie toutes les lignes en dessous
    for k in range(j+1,n) :
      alpha = A[k,j] / A[j,j]
      A[k,:] += -alpha*A[j,:]
      b[k] += -alpha*b[j]
return(A,b)
```

Avec comme fonction `chNonNul` :

```
def chNonNul :
   """
   entrée: A = array
             = matrice carrée inversible de taille nxn
           j = entier
             = indice de ligne
   sortie: k = indice de ligne
   on cherche k >= j tel que A[k,j] !=0
   """
   n, m = shape(A)
   for k in range(j,n) :
      if A[k,j] != 0 :
         return(k)
   # normalement la suite est du code mort
   # car  A est inversible
```

```
    print("ERREUR matrice non inversible")
    return()
```

Très souvent, plutôt que de chercher un terme non nul, on va chercher le pivot le plus grand (en valeur absolue). Cela assure une meilleure stabilité numérique. On parle de **pivot partiel**.

On va donc utiliser :

```
def chNonNul(A,j) :
    [n,m] = shape(A)
    pivotMax = abs(A[j,j])
    lPivot = j
    for k in range(j+1,n) :
        if abs(A[k,j]) > pivotMax :
            pivotMax = abs(A[k,j])
            lPivot = k
    return lPivot
```

V.7 Autour des polynômes

Les polynômes sont une liste de coefficients. **Attention :** si le degré du polynôme est n, alors la taille de la liste est $n+1$.

⋆ Multiplication de deux polynômes

Si $P = \sum_{k=0}^{n-1} a_k X^k$ et $Q = \sum_{l=0}^{m-1} b_l X^l$, alors le produit est obtenu avec la formule :

$$PQ = \sum_{(k,l)\in[\![0,n-1]\!]\times[\![0,m-1]\!]} a_k b_l X^{k+l},$$

ce qui signifie que les coefficients du produit se calcule ainsi : on fait les $n \times m$ produits $a_k b_l$, que l'on regroupe selon la valeur de $k+l$.

```
def produitPoly(P,Q) :
    """
    entrée: P = liste de réels
              = coefficients du polynômes P
            Q = liste de réels
              = coefficients du polynômes Q
    sortie: PQ = liste de réels
               = coefficients du polynômes PQ
    """
    n = len(P); m = len(Q);
    # attention P est de degré n-1 et Q de degré m-1

    PQ = [0]*(n+m-2) #initialisation
    for k in range(n) :
        for l in range(m) :
            PQ[k+l] += a[k]*b[l]
    return PQ
```

✯ Évaluation d'un polynôme en un point

Pour évaluer un polynôme en un point, on utilise la méthode de Hörner. Par exemple, pour calculer :

$$P(x) = a_4x^4 + a_3x^3 + a_2x^2 + a_1 + a_0 \text{ on fait } P(x) = (((a_4x + a_3)x + a_2)x + a_1)x + a_0$$

De cette manière :

- On limite le nombre d'opérations (surtout que pour le processeur l'opération $y = ax + b$ est une seule opération).
- On limite la propagation des erreurs d'arrondis.

Voici un exemple de programme :

```
def evalue(P, x):
    """
    entrée: P = liste de réels
              = coefficients du polynômes P
            P = [a0, .... , an] = sum ak X^k
            x = float
              = valeur où on évalue le polynôme
    sortie: S = float
              = valeur de P(x)
    """
    S = P[-1]
    for k in range(2,len(P) +1):
        S = (S*x +P[-k])
    return S
```

(!) Attention à la convention : le polynôme $P = \sum_{k=0}^{n} a_k X^k$ est stocké ici sous la forme de la liste $[a_0, \ldots, a_n]$ (de longueur $n+1$ donc). Il existe des convention différentes

V.8 Méthode d'Euler

On cherche une fonction y solution sur $[0,1]$ d'une équation différentielle :

$$y' = f(y,t) \qquad \text{avec} \qquad y(0) = y_0 \text{ donné.}$$

La fonction f est une fonction définie sur $J \times [0,1]$ et on suppose que $y : [0,1] \to J$. l'équation peut ainsi s'écrire :

$$\forall t \in [0,1],\ y'(t) = f\Big(y(t),t\Big) \qquad \text{avec} \qquad y(0) = y_0 \text{ donné.}$$

(R) Souvent la fonction f ne dépends pas de t (équation différentielle autonome).
Il faut savoir adapter les méthodes dans le cas où on résout sur un intervalle $[t_0,t_1]$ et non $[0,1]$

La fonction y peut être à valeurs dans $\mathbb{R}$ (on suit alors l'évolution d'une quantité physique) ou dans $\mathbb{R}^2$ ou $\mathbb{R}^3$ (on suit alors l'évolution d'un point mobile). Ce sera en particulier le cas lorsqu'on va étudier des équations d'ordre 2.

Physiquement, on connaît la situation à $t = 0$ et on souhaite déterminer l'évolution au cours du temps.

✯ Principes généraux

On **discrétise** alors l'intervalle $[0,1]$ en $n+1$ valeurs séparées par un **pas** de largeur $\Delta = \frac{1}{n}$.

Autrement dit, on considère les points $(t_i)_{i\in[\![0,n]\!]}$ définis par :

$$\forall i \in [\![0,n]\!],\ t_i = \frac{i}{n} = i\Delta.$$

! Attention : parfois on a n points et donc $n-1$ intervalles.

Si on résout sur un segment $[a,b]$, alors la discrétisation est :

$$a = t_0 < t_1 < \ldots, < t_n = b \quad \text{avec} \quad t_i = a + i\Delta = a + i\frac{(b-a)}{n}.$$

On peut obtenir rapidement une liste des valeurs $(t_i)_{i\in[\![0,n]\!]}$ par la fonction `linspace` :

```
t = linspace(a,b, nbrPoints)
```

qui renvoie un `1d-array` de taille `nbrPoints` qui contient les valeurs discrétisées de l'intervalle $[a,b]$. Les bornes a et b étant comprises.

On cherche alors à construire la liste Y des valeurs approchées de $\Big(y(t_i)\Big)_{i\in[\![0,n]\!]}$. On connaît la première valeur : $Y_0 = y(0) = y_0$.

R On utilise cette convention : la vraie solution (une fonction inconnue) est noté en minuscule y et les valeurs calculés sont notées en majuscule Y.

On utilise alors le principe suivant :

$$\begin{aligned}\forall i \in [\![0,n-1]\!],\ Y_{i+1} \approx y(t_{i+1}) =& y(t_i) + \int_{t_i}^{t_{i+1}} y'(u)du \\ =& y(t_i) + \int_{t_i}^{t_{i+1}} f(y(u),u)du\end{aligned}$$

Or cette dernière intégrale est sur un intervalle de longueur Δ, on peut donc appliquer une méthode simple pour approcher cette intégrale.

Il y a donc un lien très fort entre méthode de calcul approché des intégrales et méthodes de résolution approché des équations différentielles.

✯ Méthode d'Euler explicite

La **méthode d'Euler** (ou Euler explicite) consiste à écrire par la méthode des rectangles à gauche pour approcher cette intégrale :

$$\int_{t_i}^{t_{i+1}} f(y(u),u)du = \Delta f\Big(y(t_i),t_i\Big)$$

En remplaçant $y(t_i)$ par la valeur approchée Y_i, on justifie l'algorithme :

$Y_0 = y_0$ la valeur donnée
$\forall i \in [\![0, n-1]\!]$, $Y_{i+1} = Y_i + \Delta f(Y_i, t_i)$

On peut aussi retrouver cet algorithme en écrivant :

$$\begin{aligned} y'(t_i) &= f(y(t_i), t_i) \\ \text{et donc : } \frac{Y_{i+1} - Y_i}{\Delta} &\approx f(Y_i, t_i) \\ \text{d'où } Y_{i+1} &= Y_i + \Delta f(Y_i, t_i) \end{aligned}$$

À partir de là, il faut étudier mathématiquement la suite des valeurs calculés pour vérifier que cette suite de valeurs converge vers la vraie solution.
La notion de convergence est ici à préciser : lorsqu'on augmente le nombre de points n, on diminue le pas Δ, on calcule plus de valeurs. De plus, la vraie solution est inconnue.

■ **Exemple V.2** Pour l'équation différentielle $y' = y$ et $y(0) = 1$ sur $[0, 1]$, on construit alors la suite $(Y_i)_{i \in \mathbb{N}}$ définie par :

$Y_0 = 1$
$\forall i \in \mathbb{N}$, $Y_{i+1} = Y_i + hY_i = (1+h)Y_i$

où on a noté $h = \frac{1}{n}$ le pas. ■

De nombreuses suites de la forme $u_{n+1} = f(u_n)$ proviennent de la méthode d'Euler. Il faut savoir passer de l'équation différentielle à la suite associée par la méthode d'Euler.

✯ Implémentation

Voici un exemple d'implémentation dans le cas général :

```
def euler(f, y0, n) :
  """
  entrée: f = fct RxR --> R
          y0 = float = condition initiale
          n = nbr d'intervalles
  sortie: Y = array1d de n+1 valeurs

  Y est une  approximation de la solution
  de l'équation y'=f(y,t) sur [0,1]
  Y[i] est la valeur approchée de
  y(ti) où ti = i Delta.
  """
  Y = [0]*n
  t = linspace(0,1,n+1)
  Y[0] = y0
  Delta = 1 / n
  for i in range(n):
    Y[i+1] = Y[i] + Delta * f(Y[i], t[i])
  return Y
```

R Le même code fonctionne dans le cas d'une équation vectoriel, c'est-à-dire si $y : \mathbb{R} \to \mathbb{R}^k$ avec $f : \mathbb{R}^k \times \mathbb{R} \to \mathbb{R}^k$ à condition que l'on utilise des array pour que l'instruction `Y[i] + Delta * f(Y[i], t[i])` ait bien le sens `array+floatxarray`.

✯ Euler implicite

On peut approximer différemment l'intégrale. Par exemple, on peut choisir les rectangles à droite :

$$\int_{t_i}^{t_{i+1}} f(y(u),u)du = \Delta f\Big(y(t_{i+1}),t_{i+1}\Big)$$

On arrive alors à une équation donc Y_{i+1} est solution :

$$\begin{aligned}\forall i \in [\![0,n-1]\!],\ Y_{i+1} \approx y(t_{i+1}) =& y(t_i) + \int_{t_i}^{t_{i+1}} y'(u)du \\ =& y(t_i) + \int_{t_i}^{t_{i+1}} f(y(u),u)du \\ =& Y_i + \Delta f(Y_{i+1},t_{i+1})\end{aligned}$$

Il faut résoudre cette équation pour obtenir la valeur de Y_{i+1}.

Le schéma **d'Euler implicite** (ou Euler rétrograde) consiste à calculer les valeurs de $\Big(Y_i\Big)$ en utilisant la formule de récurrence :

$Y_0 = y_0$ la valeur donnée
$\forall i \in [\![0,n-2]\!],\ Y_{i+1} = Y_i + \Delta f(Y_{i+1},t_i)$

Il faut donc calculer la solution d'une équation à chaque étape.

On peut voir aussi cela de la manière suivante :

$$\begin{aligned} y'(t_{i+1}) &= f(y(t_{i+1}),t_{i+1}) \\ \text{donc } \frac{Y_{i+1}-Y_i}{\Delta} &\approx f\left(Y_{i+1},t_{i+1}\right) \\ \text{ce qui donne } Y_{i+1} &= Y_i + \Delta f\left(Y_{i+1},t_{i+1}\right).\end{aligned}$$

R Il existe beaucoup d'autres variantes !
On utilise souvent la méthode de Newton pour calculer la solution !

■ **Exemple V.3** Toujours pour le problème de l'exponentiel :

$$y' = y \qquad y(0) = 1.$$

La relation de récurrence est alors :

$$Y_0 = 0$$

$$\forall i \in [\![0,n-1]\!],\ Y_{i+1} = Y_i + hY_{i+1} \text{ ie } Y_{i+1} = \frac{1}{1-h}Y_i$$

où on a noté $h = \frac{1}{n}$ le pas. ■

Selon les équations la méthode d'Euler implicite est plus précise que la méthode d'Euler explicite.

✯ Cas d'une équation d'ordre 2

Pour résoudre une équation différentielles d'ordre 2, on effectue un changement d'inconnue : on considère que la fonction et sa dérivée sont inconnues. On se ramène ainsi à une équation d'ordre 1, sauf que l'inconnue est une fonction à valeur dans $\mathbb{R}^2$.

Pour formaliser le problème : On cherche une fonction y solution sur $[0,1]$ d'une équation différentielle :

$$(E) \quad y'' = f(y', y, t) \qquad \text{avec} \qquad y(0) = y_0 \text{ et } y'(0) = v_0 \text{ donnés.}$$

La fonction f est une fonction définie sur $J \times J \times [0,1]$ et on suppose que $y : [0,1] \to J$. l'équation peut ainsi s'écrire :

$$\forall t \in [0,1],\ y'(t) = f\Big(y'(t), y(t), t\Big) \qquad \text{avec} \qquad y(0) = y_0 \text{ et } y'(0) = v_0 \text{ donnés.}$$

On considère alors comme inconnue la fonction x à valeur dans J^2,

$$x :\ t \mapsto \big(y(t), y'(t)\big)$$

Physiquement, le système est déterminée par la position y et la vitesse y'. On prends donc comme inconnue ces deux valeurs.

On écrit alors l'équation vérifiée par x

$$(\tilde{E}) \quad x' = F(x,t) \qquad \text{avec} \qquad x(0) = (y_0, v_0)$$

où F est une fonction définie par :

$$F : \begin{cases} J^2 \times [0,1] & \to & J^2 \\ ((a,b),c) & \longmapsto & \Big(b, f(a,b,c)\Big) \end{cases}$$

Le rapport entre x et la solution y cherché est : y est solution de l'équation (E) si et seulement si (y, y') est solution de $\tilde{E}$.

On peut donc utiliser les techniques d'approximation valable pour le premier ordre sur cette équation, trouver la valeur approchée de x, et ne garder que la première composante.

On retiendra la méthode : dans le cas d'une équation différentielle d'ordre 2, on se ramène à l'ordre 1 en prenant comme inconnue la fonction et sa dérivée.

■ **Exemple V.4** Si on étudie l'équation du pendule amorti :

$$(E) \qquad \theta'' + \lambda\theta' + \omega^2 \sin(\theta) = 0$$

le pendule est lâché sans vitesse initiale ($\theta'(0) = 0$) et à l'angle θ_0.

On va considérer la fonction y à valeur dans $\mathbb{R}^2$, solution de :

$$(E_2) \qquad y' = f(y) \text{ avec } f : (a,b) \longmapsto (b, -\lambda b + \omega^2 \sin(a)).$$

On a alors :

$$\theta \text{ est solution de } (E) \iff (\theta, \theta') \text{ est solution de } (E_2).$$

On applique ensuite les méthodes de résolution numériques des équations différentielles sur l'équation (E_2).

Par exemple en utilisant la méthode d'Euler, on peut utiliser les opérations entre `array`.

Pour définir la fonction f :

```
def f(u) :
  """
  entrée: u = array1d de longueur 2
  sortie: v = array1d de longueur 2
  """
  a, b = u
  return array( [b, -LAMBDA*b + omega**2*sin(a)] )
```

On choisit la sortie sous forme d'array.

Pour la résolution, on met la solution dans un 2d array : la ligne i contient les valeurs de $(\theta(t_i), \theta'(t_i))$. Cela donne :

```
def Resolution(theta0, n):
  """
  entrée: n = nbr d'intervalles
          theta0 = angle initial
 ,sortie: Y = 2d array de taille (n+1) x 2
          Y[i,0] = valeur approchée de theta au tps i
          Y[i,0] = valeur approchée de theta' au tps i
  """
  Y = zeros ((n+1,2))
  Y[0,0] =  theta0 # angle initial
  delta = 1/n
  for i in range(n) :
    Y[i+1,:] = Y[i,:] + delta*f( Y[i,:])
```

L'utilisation de `array` permet d'utiliser directement la formule : $Y_{i+1} = Y_i + \Delta f(Y_i)$ puisque l'on faire des opérations entre array (et donc des additions vectorielles). ∎

✯ Généralisation

Les **méthodes aux différences finies** consistent à remplacer des dérivées par des différences entre valeur proche.

Si on note $\left(t_i\right)_{i\in[\![0,n]\!]}$ la discrétisation, on a alors vu les approximations de la première dérivée par la méthode d'Euler explicite :

$$\begin{aligned} f'(t_i) &\approx \frac{f(t_{i+1}) - f(t_i)}{t_{i+1} - t_i} \\ &\approx \frac{f(t_{i+1}) - f(t_i)}{\Delta} \end{aligned}$$

et la méthode d'Euler implicite :

$$\begin{aligned} f'(t_{i+1}) &\approx \frac{f(t_{i+1})-f(t_i)}{t_{i+1}-t_i} \\ &\approx \frac{f(t_{i+1})-f(t_i)}{\Delta} \end{aligned}$$

On peur ajouter la méthode centrée :

$$\begin{aligned} f'(t_i) &\approx \frac{f(t_{i+1})-f(t_{i-1})}{t_{i+1}-t_{i-1}} \\ &\approx \frac{f(t_{i+1})-f(t_i)}{2\Delta} \end{aligned}$$

Pour la dérivée seconde, on peut utiliser :

$$f''(t_i) \approx \frac{f(t_{i+1})-2f(t_i)+f(t_{i-1})}{\Delta^2}$$

de plus ces approximations sont aussi valables pour une fonction de deux variables. Par, exemple, si $f : \mathbb{R}^2 \to \mathbb{R}$ est une fonction de (x,t), alors :

$$\frac{\partial^2 f}{\partial t^2}(x_j,t_i) \approx \frac{f(x_j,t_{i+1})-2f(x_j,t_i)+f(x_j,t_{i-1})}{2\Delta^2}$$

C'est les formules de Taylor qui justifie cette idée, en particulier pour l'ordre 2, on a :

$$\begin{aligned} f(t_{i+1}) &= f(t_i)+\Delta\frac{\partial f}{\partial t}(t_i)+\Delta^2\frac{\partial^2 f}{\partial t^2}(t_i)+O\left(\Delta^3\right) \\ \text{et } f(t_{i-1}) &= f(t_i)-\Delta\frac{\partial f}{\partial t}(t_i)+\Delta^2\frac{\partial^2 f}{\partial t^2}(t_i)+O\left(\Delta^3\right) \\ \text{ainsi } f(t_{i+1})-2f(t_i)+f(t_{i-1}) &= 2\Delta^2\frac{\partial^2 f}{\partial t^2}(t_i)+O(\Delta^3). \end{aligned}$$

On dit que c'est une approximation d'ordre 3.

V.9 Autour des nombes premiers

On souhaite construire la liste des nombres premiers inférieurs ou égaux à n en utilisant la méthode dite du crible d'Ératosthène.

On commence par construire une liste de booléens notée `listeBool` de longueur `n` telle que :
`listeBool[i]=True` si i est premier. Pour cela :

- On commence par mettre toutes les valeurs à `True` (*i.e.* tous les nombres sont *a priori* premiers)
- On initialise ensuite les termes 0 et 1 à `False`,
- 2 est premier : on enlève alors tous les multiples de 2, en posant `listeBool[j]=False` pour toutes les valeurs de j qui s'écrivent $j = 2k$ avec $k > 1$.
- 3 est premier : on enlève alors tous les multiples de 3, etc.
- pour chaque i qui n'a pas été enlevé précédemment, on enlève les multiples de i (sauf i lui-même).

Dans une deuxième étape, on reconstruit la liste des nombres premiers par concaténation à partir de `listeBool`

Écrire une fonction `tablePremier` qui prend en entrée un entier `n` et qui construit la liste des nombres premiers inférieurs ou égaux à n.

Indications : pour déterminer les multiples de i, il faut penser à la boucle `for` avec un pas de `i`. Pour initialiser `listeBool` on peut utiliser la répétition : `listeBool = [True]*n`.

```
def tablePremier(nFinal) :
    """
    entrée: nFinal = entier
    sortie: listePrem = liste d'entiers
    construit la liste des entiers premiers <= nFinal
    """
    #étape 1: on va construire une liste listeBool True/False
    #avec listBool[i] = True si i est premier
    listeBool =[True]*(nFinal+1) #on intialise avec que des True

    listeBool[0] = False # 0 n'est pas premier
    listeBool[1] = False # 1 n'est pas premier

    for i in range(2,nFinal+1) :
        #si i n'est pas premier, il n'y a rien à faire
        if listeBool[i] :
            #on efface tous les multiples de i:
            for j in range(i*i,nFinal+1,i) :
                listeBool[j] = False

    #étape 2: à partir de listeBool, on construit listePrem
    #par concaténation
    listePrem = []
    for i in range(nFinal+1) :
        if listeBool[i] :
            listePrem += [ i ]

    return(listePrem)
```

Généralités et syntaxe des structures de contrôles

Pelletier Sylvain PSI, LMSC

⋆ Généralités, vocabulaire

Chemin d'accès aux fichiers :	`data/img.jpeg` = fichier `img.jpeg` dans le dossier `data`
PAS D'ACCENTS, PAS D'ESPACE DANS LES NOMS DE FICHIERS	
RAM :	Mémoire utilisable par le processeur
Processeur :	Unité de calcul de l'ordinateur, rapide mais très peu évoluée.
Instruction compréhensible pour le processeur :	Le processeur va lire dans la RAM deux nombres et l'opération demandée. Il effectue l'opération et copie le résultat dans la RAM.
Interpréteur Python :	Programme qui va lire une phrase écrite dans un langage proche du langage courant (le langage de programmation Python) et transforme cette phrase en instructions compréhensibles pour le processeur.
Script python :	Fichier texte contenant des commandes. L'interpréteur lit le script ligne par ligne, et exécute les commandes au fur et à mesure.
Variable :	Symbole désignant une partie de la RAM dans laquelle est stockée un nombre, une chaîne de caractères, etc. Les variables servent à stocker les calculs effectués.
Affectation de variable :	La variable est affectée si elle est à gauche du signe égal. Son contenu est alors créé ou modifié. Une variable doit être affectée avant d'être utilisée.
Nom des variables :	Le nom de variables doit commencer par une lettre, peut contenir un chiffre ou un `_`, mais pas d'espace, pas d'accents, pas de point. Sensible aux majuscules/minuscules.
Expression :	Résultat d'un calcul. Toute instruction doit être de la forme : `variable = expression` . L'expression est évaluée, et si elle a un sens, la valeur calculée est affectée à la variable. **NB :** si l'expression n'est pas affectée à une variable, alors le calcul est effectué, puis détruit.

⋆ Le langage Python

Indentation :	Nombre d'espaces en début de ligne
Bloc d'instructions :	Plusieurs instructions délimitées par l'indentation et le :
Instructions sur plusieurs lignes :	Utiliser des parenthèses ou \
Plusieurs instructions sur une seule ligne :	Séparer les instructions par ;
Commentaire sur une ligne :	`#` (`ALTGR+3`), permet de donner des explications et d'effacer rapidement une partie du code
Commentaires sur plusieurs lignes :	Texte délimité par `"""` (« doc string »)

✯ Les raccourcis clavier pour Spyder

Arrêter l'interpréteur si boucle infinie :	`CTRL+C` ou `Moniteur système`
Quitter l'interpréteur :	`CTRL+D` ou `quit()`. Ne pas fermer la fenêtre.
Copier / Coller / Couper :	`CTRL+C` / `CTRL+X`/ `CTRL+V` ou menu `Fichier`
Annuler :	`CTRL+Z`. Utile si erreur de manipulation.
Rechercher / Remplacer :	`CTRL+F` / `CTRL+H` ou menu `Recherche`
Compléter :	`CTRL+ESPACE`
Exécuter :	`F5` ou menu `Exécution`
Options d'exécution :	Si possible : « *Exécuter dans un terminal système externe* + *Interagir avec l'interpréteur Python après exécution* » Sinon :« *Exécuter dans un interpréteur dédié* + *Interagir avec l'interpréteur Python après exécution* ». Si erreur, utilisez `F6` ou `CTRL+F6` ou menu `Exécution-> configurer`
Commenter / Décommenter :	`CTRL+1` ou menu `Fichier`
Indenter (une région ou une ligne) :	touche `Tab`, ne pas utiliser `espace`

✯ Les raccourcis clavier pour Pyzo

Arrêter l'interpréteur si boucle infinie :	cliquer sur la croix rouge. Il faut alors ouvrir un nouveau shell
Copier / Coller / Couper :	`CTRL+C` / `CTRL+X`/ `CTRL+V` ou menu `Fichier`
Annuler :	`CTRL+Z`. Utile si erreur de manipulation.
Rechercher / Remplacer :	`CTRL+F` / `CTRL+H` ou menu `Recherche`
Compléter :	lorsqu'il y a une info-bulle, utilisez `TAB`
Exécuter :	`CTRL+Shift+E` ou menu `Exécuter`
Commenter / Décommenter :	`CTRL+R` ou menu `Édition`
Indenter (une région ou une ligne) :	touche `Tab`, ne pas utiliser `espace`

Utiliser `CTRL+Shift+E` ou `CTRL+F5` et non `CTRL+E` ou `F5` ! Sinon le fichier est exécuté à la suite du fichier précédent.

✯ Les types de variables simples et les opérations

Types	Opérations	Commentaires

Les booléens	`B1 or B2`
	`B1 and B2`
	`not B`
		Résultat de test, valeur `True` ou `False`. On les utilise pour les `if` et les `while`.

Les chaînes de caractères	Concaténation : `s1 + s2` Comparaison : `s1 == s2` Ordre alphabétique : `s1 >s2` Répétition : `int * s`	Sortie de input, mots entre `"`
Les entiers	Opérations usuelles : `+,-,*, /` Comparaison : `==, >` Puissance : `**` Quotient division euclidienne : `//` Reste division euclidienne : `%`	Calcul exact. La division de deux entiers donne un flottant. Conversion automatique en flottant si besoin.
Les flottants	Opérations usuelles : `+,-,*, /` Comparaison : `==, >` Puissance : `**`	Calcul approché. Ne pas utiliser `==` entre deux flottants.

✯ Conversion, affichage

Pour convertir : `int`, `float`, `str`.
Pour afficher : `print("la valeur de x est "+str(x))`
Caractères spéciaux :

- `\n` = changement de ligne
- `\t` = tabulation (déplace vers la droite)
- `\r` = retour chariot.

- \" = les guillemets (ne pas mettre de " dans une chaîne de caractères).
- \\ = le caractère \.

R Les caractères spéciaux compte pour 1 caractère.

On peut utiliser ", ' pour délimiter une chaîne de caractère.

Dans une chaîne de caractères délimitée par ", on peut utiliser '. Au contraire, dans une chaîne de caractères délimitée par ', on peut utiliser ". On peut aussi utiliser """ ce qui permet de mettre des chaînes de caractères sur plusieurs lignes docstring.

■ **Exemple V.5** Voici comment délimiter des chaînes de caractères :

```
msg1 = "C'est bon"
msg2 = 'il a dit: "ok"'
msg3 = """grande nouvelle:
on peut passer à la ligne
dans une chaîne de caractères"""
```

■

✯ La structure conditionnelle

On peut exécuter certaines commandes si la valeur d'un booléen / d'un test est `True` :

```
if test1 :
    bloc si test1 est vrai
elif test2 :
    bloc optionnel si test1 est faux et test2 est vrai
elif test3 :
    bloc optionnel si test1 est faux et test2 est faux et test3 est vrai
else :
    bloc optionnel cas restants
```

Tests possibles	== (égalité), >, <, >=, <=. On peut aussi utiliser `if B:` avec B une variable booléenne.
Évaluation paresseuse	Si la case existe ET qu'elle contient 0

- On peut enchaîner les tests : `if a<b<c`. Cela est équivalent à `if a<b and b<c`.

✯ La structure répétitive, `for`

La boucle `for` permet de répéter un bloc d'instructions :

```
for i in range(n):
    bloc d instructions répétés n fois
suite des commandes (non répétées)
```

- `i` est une variable, `n` une expression,
- `i` est initialisée (à 0) et incrémentée à chaque passage (de 1), jusqu'à la valeur finale non atteinte ($n-1$).
- On sait à l'avance combien de fois le bloc va être répété.
- Ne pas utiliser `i` en dehors de la boucle et ne pas modifier la valeur de `n` dans la boucle.

Balayage d'un tableau	`for i in range(n):`
	`  for j in range(m):`
	`  traitement de la case (i,j)`

Fonction range générale :	`for i in range(start,stop, step):` en particulier, `step` peut-être négatif. La valeur finale n'est jamais atteinte. Une boucle vide n'est pas interprétée. `step` doit être un entier.

✯ La structure répétitive, `while`

La boucle `while` permet de répéter un bloc d'instructions tant qu'une condition est respectée

```
while test :
    bloc d instructions répététant que test est vrai
suite de l instruction lorsque test est faux
```

- On ne sait pas à l'avance combien de fois le bloc doit être répété.

- Toujours écrire au brouillon la négation de la condition (« jusqu'à ce que »). Lorsque l'on sort de la boucle, le test est faux.
- Si le `test` est faux au premier passage, le bloc n'est jamais exécuté.
- Si le `while` s'écrit `P et Q` , alors il est souvent suivi d'un `if` qui permet de savoir quelle condition a fait sortir de la boucle.
- Le `test` peut passer à `False` dans la boucle. La boucle ne se finit que quand le test est `False` à la fin du bloc d'instructions.
- Lorsque la boucle est finie, le test est faux.

Fonctions, bibliothèques

Pelletier Sylvain PSI, LMSC

⋆ Fonctions

Une **fonction** est une portion de code effectuant une tâche ou un calcul à partir de valeurs en **entrées**.

```
def nomFonction(x1,x2):
    """
    entrée: x1 = type = interprétation
            x2 = type = interprétation
    sortie: ....
    """
    #   bloc d'instructions permettant de calculer les sorties,
    #   les  variables x1,x2 étant supposées connues.
    return y1,y2,y3

#code permettant l'utilisation de la fonction:
a,b,c = nomFonction(u,v)
```

- Le nom des variables utilisées pour définir la fonction (ici `x1,x2`) ne correspondent pas nécessairement au nom des expressions utilisées pour appeler la fonction (ici `u,v`).
- On peut faire `return expression`
- Une fonction peut contenir plusieurs `return`. Elle se termine au premier `return` évalué.
  Utile pour gagner du temps de calcul :

```
def listeEgale(l1,l2):
    if len(l1) != len(l2):
        return False
    for i in range(len(l1)):
        if l1[i]!=l2[i]:
            return False
    return True
```

  une fonction peut ne rien renvoyer, en fait elle renvoie `None`. On peut aussi mettre `return` pour envoyer `None`.
- Une fonction est une variable comme les autres : Si on définit la fonction `f` alors on ne peut pas utiliser une autre variable `f` et une fonction peut être une entrée d'une autre fonction : `def dichotomie(f,a,b,eps):`.
- Convention : des noms d'action pour les fonctions, des noms d'objets pour les variables.

⋆ Bibliothèques

Une **bibliothèque** est un fichier contenant des fonctions utilisables dans plusieurs projets.

- On dispose donc de deux fichiers : le module `prof.py` contenant les fonctions `f,g` et `h`, et le script `test.py` contenant des instructions.
- Il faut placer le fichier `prof.py` dans le même répertoire que `test.py`. Pour les bibliothèques scientifiques, elles sont placées automatiquement dans un répertoire spécial.
- Dans `test.py`, on ajoute la ligne :
  `from prof import f,g,h`
- les fonctions `f,g`, et `h` sont alors utilisables dans `test.py`.
- Les bibliothèques standards que l'on utilise souvent sont `numpy`, `scipy`, `matplotlib`, `pylab`. On aura donc souvent dans les fonctions importées :
  `from pylab import shape, ones`

⋆ Quelques fonctions de la bibliothèque pylab

Voici une liste des fonctions que l'on rencontre le plus souvent. Pour connaître la syntaxe d'une fonction il faut chercher dans l'aide ou sur internet. Toutes ces fonctions se trouvent dans le module `pylab`. Plus précisément, le module pylab regroupe les modules :

math qui contient les fonctions mathématiques,
numpy, scipy qui contient les module de calculs numériques et scientifiques (structure de tableau, de matrices)
random qui contient les fonctions pour l'aléatoire.
matplotlib qui contient les fonctions graphiques.

PIL qui contient la manipulation des images

Dans le tableaux suivants :

- `x,y` désigne des réels
- `n,m` désigne des entiers.
- `L` désigne une liste.

`sin(x)`, `cos(x)`, `tan(x)`	Les fonctions trigonométriques
`acos(x)`, `asin(x)`,`atan(x)`	Les fonctions trigonométriques réciproques
`exp(x)`, `log(x)`,`log10(x)`	Les fonctions exponentiel, logarithme népérien et logarithme en base 10
`pi`, `e`	constante mathématiques
`sqrt(x)`	la fonction racine
`abs(x)`	la fonction valeur absolue
`floor(x)`, `ceil(x)`, `round(x)`	la valeur approchée (par défaut, excès, la plus proche)
`max(x,y)`, `max(L)` `min(x,y)`, `min(L)`	le maximum, minimum
`fsum(L)`	somme des éléments d'une liste
`array(LL)`	transforme une liste de liste en tableau
`mat(LL)`	transforme une liste de liste en tableau
`rand()`	aléatoire dans $[0,1]$
`randint(n,m)`	aléatoire dans $[\![n,m]\!]$
`plot(x,y)`	dessine un trait
`imshow(tab)`	montre une image en niveaux de gris correspondant à `tab`

Les listes et les tableaux

Pelletier Sylvain PSI, LMSC

⋆ Manipulation des listes (les fondamentaux)

Liste :	Structure qui contient un certain nombre d'éléments, dans un certain ordre. On peut ajouter et supprimer des éléments n'importe où dans la séquence. Une liste peut contenir différent type de données et même des listes.
Création de liste :	On utilise les crochets, les éléments sont séparés par une virgule. On peut créer une liste vide.
Accès aux éléments :	`len(L)` donne la longueur de la liste. On accède à l'élément `k` par `l[k]`, pour $k \in [\![0, n-1]\!]$
Ajout d'un élément :	On peut faire `L.append(a)` pour ajouter `a` à la fin de `L`.
Dépiler un élément :	Pour la structure de pile, on peut faire `y=L.pop()` qui enlève le dernier élément de `L` et qui affecte sa valeur à y.
Concaténation :	`L1+L2` concatène les liste `L1` et `L2`. **Attention :** le `+` est entre élément de même type (liste ou liste de listes, etc). Par exemple, `L += [a]` est équivalent à `L.append(a)`. On peut aussi utiliser `L1.extend(L2)`
Répétition :	En utilisant `*` on peut répéter plusieurs fois une liste. Utile pour initialiser une liste. Exemple : `L = [0]*n` : liste de longueur n qui ne contient que des 0.
Effacer dans une liste :	On efface l'élément i d'une liste avec la commande `del(L[i])`. La taille de la liste est alors automatiquement diminuée de 1.

Parcours d'une liste :	Boucle sur les éléments :
	`for x in L:`
	`  boucle sur x élément de L`
	Boucle sur la position de lecture :
	`for i in range(len(L)):`
	`  boucle sur i, on doit utiliser L[i]`
	Boucle sur les deux :
	`for i,x in enumerate(liste):`
	`  boucle sur x et i`
	lorsque l'on a besoin de la position de lecture, on utilise `for i in range(len(L))`
	lorsque l'on a uniquement besoin de l'élément, on utilise `for x in L`

Appartenance :	`0 in liste` renvoie `True` si et seulement si 0 est dans la liste. La négation s'écrit `0 not in liste`, qui signifie donc « 0 n'est pas dans la liste ».
Égalité :	`L1 == L2` renvoie `True` si et seulement si `L1` et `L2` sont identiques.
Liste en compréhension :	`[exp for variable in liste if condition]`

⋆ Manipulation des listes (les détails)

Slicing (extraction) :	`liste[start:stop]` extrait de liste les éléments de start à stop (exclu). On peut aussi faire : `liste[:stop]` (`start=0`), `liste[start:]` (`stop=len(liste)`) `liste[:]`(tous pour faire la copie), `liste[start:stop:step]`.
Détails sur le Slicing :	Si l'extraction est vide (`start>=stop`) on a une liste vide. Si `stop` dépasse le maximum d'éléments, il n'y a pas d'erreurs.
Application du slicing	l'opération élémentaire $l_i \leftarrow l_i + a l_j$ s'écrit : `mat[i,:] += a*mat[j,:]` insertion de x dans L en position i `L = L[:i] + [x] + L[i:]`
Indexation négative :	`L[-1]` donne le dernier élément, `L[-2]` l'avant dernier, etc.
Exemple :	on veut savoir si le nom du fichier finit par `.pdf`, on écrit alors : `nomFichier[-4:] ==".pdf"`.

⋆ Liste de liste

Une liste `L` peut contenir des listes de tailles quelconques. C'est utile pour :

- manipuler des ensembles d'ensemble,
- faire des tableaux (bien que la structure de array soit plus adapté),
- manipuler des structures de données complexes, lorsque par exemple on a plusieurs objets.

Liste de liste :	L'accès à l'élément (i,j) est `L[i][j]`. La longueur de la liste `L[i]` est `len(L[i])`.
Unpacking :	`a, b, c= L`, ou d'une manière générale : `variables séparées par virgule=liste`. Il doit y avoir autant de variables que d'éléments dans `listes`.
Utilisation de l'unpacking :	Initialiser plusieurs variables par affectation simultanée : `a,b,c=1,2,3`. En particulier, échange de deux variables : `a,b=b,a`. Boucler sur des listes de listes : `for ii,jj in listePixel:` Récupérer la sortie d'une fonction qui renvoie plusieurs valeurs : `n,m =tailleImage(img)`.

⋆ Tableaux

Pour stocker un tableau d'éléments de même nature, et de taille fixée on utilise un array et non une liste de liste. La structure d'array est dans la bibliothèque `pylab`.

On ajoute donc dans la partie « fonctions importées » :

```
from pylab import ones, zeros, array, shape
```

`ones((n,m), dtype = int)` `ones((n,m), int)`	crée un tableau d'entiers de taille (n,m) contenant des 1
`zeros((n,m))`	crée un tableau de flottants de taille (n,m) contenant des 0.
`dtype` peut être `int`, `float` (par défaut), `bool`, `str`	
`array([[1,2],[3,4],[5,6]])`	transforme une liste de listes en array si possible.
Accès aux éléments :	`[n,m]=shape(tab)` donne la taille de `tab`, `tab[i,j]` permets d'accéder à l'élément (i,j).
Opération sur les tableaux :	`reel+tab` : ajoute à chaque élément de `tab` la valeur de `reel`. `reel*tab` : multiplie chaque élément de `tab` par `reel`. `tab1*tab2` multiplication élément par élément si les tableaux `tab1` et `tab2` ont la même taille.
Comparaison de tableaux :	`tab1 == tab2` renvoie `False` si `tab1` et `tab2` n'ont pas la même taille, sinon compare élément par élément et renvoie un tableau de `False` / `True`.
Copie de tableaux :	Les tableaux sont mutables. Pour les copier, il faut utiliser la fonction `copy`.

Les possibilités graphiques de la bibliothèque matplotlib

Pelletier Sylvain PSI, LMSC

Ce td est autant une liste d'exercices qu'une fiche de cours à conserver !

Le but de ce td est d'introduire les possibilités graphiques de Python. Pour cela, nous allons utiliser des librairies pour le calcul scientifique et le graphisme (dessin de fonctions, d'histogrammes, traitement des images).

La totalité des fonctions graphiques se trouvent dans la bibliothèque `matplotlib`.

Nous utiliserons pour commencer la bibliothèque `pylab` (« *Python* et *Matlab* ») qui contient les bibliothèques :

matplotlib pour le graphique donc,
numpy pour la manipulation des tableaux `array`
PIL pour la manipulation d'images

Pour ce TP, on commencera donc les scripts par l'instruction :

```
from pylab import *
```

Cette instruction est à mettre bien sûr une seule fois en haut du script.

Les fonctions graphiques (`plot`), les fonctions mathématiques (cosinus, sinus, etc.), les constantes mathématiques (`pi`, `e`) et la structure `array` sont alors disponibles.

Cette méthode d'importation de fonctions n'est pas conseillée par les spécialistes de Python, on l'utilise ici comme une première approche pour simplifier la syntaxe. Pour un oral de mathématiques informatique, c'est la technique conseillée pour aller plus vite.

Dans les prochains tds, on préférera importer chaque fonction une à une, ou importer avec un alias. La technique utilisée souvent dans les énoncés de concours est d'importer la bibliothèque `numpy` et `matplolib` avec des alias avec :

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

Il faut alors faire précéder toutes les fonctions numériques par `np` et toutes les fonctions graphiques par `plt`.

- Pour faire des graphismes complexes pour présenter des résultats scientifiques (par exemple pour les TIPE), il est souvent parfois plus simple de sauvegarder les données obtenues avec Python dans un fichier (par exemple un fichier CSV) puis de les interpréter graphiquement avec un autre logiciel.
  On peut aussi sauvegarder la figure obtenue avec Python sous forme d'image pour l'incorporer avec une légende et un titre dans un document.
- À l'oral de certains concours, on résout un problème mathématique ou physique en s'appuyant sur les résultats obtenus par l'ordinateur.
  Par exemple, on va faire tracer une courbe par l'ordinateur et en déduire les solutions de l'équation $f(x) = 0$. Il est donc important de connaître un minimum de fonctions graphiques.
  Il ne s'agit pas non plus d'avoir une connaissance encyclopédique des possibilités graphiques, mais de se débrouiller avec quelques outils simples.
  Ne pas hésiter à utiliser l'aide, avec la syntaxe `help(fct)`.

⋆ La fonction plot

La fonction la plus importante est la fonction `plot`. La syntaxe est :

```
plot(listeX, listeY) ou plot(listeX,listeY, [options] )
```

Les entrées sont deux listes (plus précisément des `array` unidimensionnels) de même taille :

- la liste des abscisses `listeX` : $(x_0, x_1, \ldots, x_{n-1})$,

- la liste des ordonnées `listeY` : $(y_0, y_1, \ldots, y_{n-1})$

Les n points

$$A_0 = (x_0, y_0),\ A_1 = (x_1, y_1),\ \ldots,\ A_{n-1} = (x_{n-1}, y_{n-1})$$

sont alors reliés par des segments de droites.

Plus précisément, un objet graphique est créé, qui peut être affiché avec la commande `show()`. L'exécution est stoppée jusqu'à ce que le graphique soit fermé.

On peut aussi créer plusieurs objets graphiques (*i.e.* plusieurs courbes) puis les afficher avec `show()`.

Quelques détails :

- Si il n'y a qu'un seul point, alors Python représente un point.
- Par défaut, Python change de couleur pour chaque courbe (bleu, vert, rouge, bleu ciel, violet). Dans la quasi-totalité des cas, cela suffit à les différencier.
- Les options concernent les couleurs et le style (trait plein pointillé), largeur de trait. Voici quelques idées :

```
plot(listeX, listeY, 'r--') # fait une ligne rouge (=r) en tiret (=--)
plot(x, y, 'bo') # fait un rond (o) bleu utile pour un point.
plot(listeX , listeY, color='red', linewidth = '2', linestyle=':')
# couleur rouge et épaisseur 2 et pointillé
```

  voir `help(plot)` pour une liste complète.
- Une fois que le graphique est ouvert, on peut modifier les axes et zoomer avec la souris. Il est toujours mieux de fixer les axes à l'avance avec les commandes :

```
axis([xmin,xmax,ymin,ymax]) # fixe la taille de la fenêtre
axis('equal') # oblige à utiliser un repère orthonormé
```

  On peut aussi utiliser la commande `grid()` qui permet d'afficher une grille.
  On peut fixer les tailles en x et y avec : `xlim(a, b)` et `ylim(a,b)`.
- Éventuellement, on peut mettre une légende au graphique et aux axes avec les instructions :

```
title(titre) # titre est un str
xlabel(labelx) # labelx est un str
ylabel(labely)
```

  Si on affiche plusieurs graphiques, il faut ajouter l'option `label` à `plot` et utiliser l'instruction `legend()`.

```
plot(listeX1, listeY1, label = "courbe 1")
plot(listeX2, listeY2, label = "courbe 2")
legend()
```

- On peut aussi sauvegarder la figure (au format png, jpeg, etc.) en cliquant sur l'icône dans la fenêtre graphique. Cela est aussi possible avec la commande `savefig(nomFig)`, avec `nomFig` une chaîne de caractères. Le format est alors donné par l'extension du fichier, *i.e.* si on sauvegarde sous le nom « img.png » le format est PNG, etc.

Exercice 1 Exécuter (et comprendre) les commandes suivantes :

```
from pylab import *
plot([1, 2, 3], [1, 3, 1])
plot([1.5, 2.5], [2, 2])
show()
```

Dessiner ensuite un « B ».

Voici un exemple avec ajout d'information :

```
from pylab import *
plot([1, 2, 3], [1, 3, 1], label= "trait vertical")
plot([1.5, 2.5], [2, 2], label = "trait horizontal")
title("une super lettre!")
legend()
show()
```

✯ Dessin de courbes et de fonctions

Il faut garder en tête que Python ne fonctionne qu'avec des vecteurs, il est donc impossible de gérer des choses complexes comme *Afficher la Courbe représentative de la fonction f*.

Il faut donc systématiquement passer par deux vecteurs (des listes ou des `array` unidimensionnels) :

- l'un contiendra une liste des abscisses $[x_0,\dots,x_{n-1}]$,
- l'autre la liste des ordonnées $[f(x_1),\dots,f(x_n)]$.

Python dessinera alors les segments de droites qui relient les n points de coordonnées $\Big((x_i,f(x_i)\Big)$. Si il y a suffisamment de points, la courbe ressemblera à une courbe lisse et non à des segments de droites.

De même pour dessiner un cercle, il faut créer la liste des coordonnées des points du cercle. On utilise alors l'équation paramétrique du cercle.

Exercice 2 Exécuter (et comprendre) les commandes suivantes :

```
from pylab import *
x = zeros(100)
y = zeros(100)
for k in range(100) :
    theta = 2*k*pi/100
    x[k] = cos(theta)
    y[k] = sin(theta)

axis([-1.2,1.2,-1.2, 1.2])
grid()
plot(x,y)
show()
```

Comment faire pour fermer le cercle ?

✯ Opérations sur les array

Le problème qui arrive alors naturellement est de créer rapidement et facilement la liste des abscisses $[x_0,\dots,x_{n-1}]$, et des ordonnées. $[f(x_0),\dots,f(x_{n-1})]$. Ces listes peuvent être des listes de réels ou des `array` unidimensionnels (en fait n'importe quelle structure que Python peut convertir en `array` unidimensionnel).

Une première méthode est, comme on l'a vu, de faire des boucles `for` :

- On crée un array avec les fonctions `ones` ou `zeros` :

```
x = ones(nbrPoints) # ou x = ones(nbrPoints, float)
```

- On remplit les valeurs dans une boucle `for` :

```
for k in range(nbrPoints) :
    x[k] = ...
    y[k] = ...
```

- Il ne reste plus qu'à envoyer ces listes à `plot`.

Une meilleure méthode est d'utiliser :

- les fonctions qui créent des listes de points équirépartis pour la liste des abscisses,
- les opérations sur les `array` pour calculer la liste des ordonnées.

Pour les abscisses, on dispose de deux fonctions à connaître :

- `linspace(xmin,xmax,nbrPoints)` (« *linéairement espacée* ») qui crée un `array` de taille `nbrPoints`, dont le premier élément est `xmin` et le dernier `xmax`.
  On contrôle ainsi le nombre de points de la discrétisation de l'intervalle $[\texttt{xmin},\texttt{xmax}]$, le pas est : $\dfrac{\texttt{xmax}-\texttt{xmin}}{\texttt{nbrPoints}-1}$.
- `arange(xmin,xmax,pas)` (« *array range* ») : crée un `array` dont le premier élément est `xmin` et dont chaque élément est distant de `pas`, le dernier étant strictement inférieur à `xmax`.
  Cette fonction est donc semblable à `range` sauf que l'on accepte ici un pas non entier, et que la sortie est un `array`.
  On contrôle ainsi le pas de la discrétisation de l'intervalle $[\texttt{xmin},\texttt{xmax}[$.
  Attention : le dernier point n'est jamais `xmax`.

R Sauf indication contraire, il est plus simple de travailler avec `linspace`. Les fonctions `linspace` et `arange` sont à connaître !

Pour les opérations sur les `array`, on rappelle que si `x` et `y` sont des `array` et `alpha` est un scalaire, on a :

- `x+y` est obtenu en ajoutant terme à terme les éléments de `x` et `y`,
- `x+alpha` est obtenu en ajoutant `alpha` à tous les termes,
- `alpha*x` est obtenu en mutlipliant tous les termes par `alpha`,
- `x*y` est obtenu en mutlipliant terme à terme les éléments de `x` et `y`,
- `cos(x)` est obtenu en appliquant la fonction cosinus à tous les éléments de `x`. C'est le comportement pour toutes les fonctions mathématiques (fonctions universelles présentes dans la bibliothèque `numpy`).

  ! Pour appliquer une fonction qui n'est pas dans la bibliothèque, il faut revenir à la boucle `for`.

- `x**n` est obtenu en mettant à la puissance n-ième tous les éléments de x.
- `1/x` est obtenu en prenant l'inverse de tous les éléments de x.

Ces fonctionnalités permettent de calculer la liste des images très facilement à partir de la liste des indices.

■ **Exemple V.6** si on doit dessiner la courbe représentative de la fonction :

$$x \longmapsto \frac{x^3+3x+2}{x^2+1} + x\sin(x) + e^x.$$

sur l'intervalle $[-10, 10]$, on écrit simplement :

```
x = linspace(-10, 10, 100)
y =  (x**3+3*x+2) / (x**2+1)  + x*sin(x) + exp(x)
plot(x,y)
show()
```

La même chose avec une boucle `for` est :

```
pas = (10 - (-10) ) / 99
x = zeros(100)
y = zeros(100)
for k in range(100):
    x[k] = -10 + k *pas
    y[k] = (x[k]**3 + 3*x[k] + 2) / (x[k]**2 +1) + x[k]*sin(x[k]) + exp(x[k])
```

■

■ **Exemple V.7** De même, le dessin du cercle peut se faire avec :

```
theta = linspace(0,2*pi,100)
x=cos(theta)
y=sin(theta)
plot(x,y)
show()
```

ou même :

```
theta = linspace(0,2*pi,100)
plot(cos(theta), sin(theta))
```

■

- Du fait des opérations entre `array`, le comportement de `+=` n'est pas celui des listes. Ainsi, l'opération `x+=1` ajoute 1 à tous les éléments de x et non à la fin. Pour ajouter une valeur, on peut utiliser la fonction `vecteur = append(vecteur, valeur)`, avec `vecteur` le `array`, et `valeur` la valeur (ou la liste de valeurs) à ajouter.
- On travaille généralement avec environ 100 points. C'est d'ailleurs la valeur par défaut pour la fonction `linspace`. Commencez par cette valeur et ajoutez des points si nécessaires.
- Les fonctions universelles s'écrivent comme les fonctions mathématiques. Attention simplement au logarithme qui s'écrit `log`. Le logarithme en base 10 s'écrit `log10`.
  Voir le tableau des fonctions universelles.
- Les opérations entre `array` sont aussi valables pour les `array` en deux dimensions (les matrices donc).
  En particulier, le produit $A*B$ n'est pas le produit matriciel, c'est le produit terme à terme. Pour obtenir le produit matriciel, il faut utiliser `dot(A,B)`.

Attention aussi : $A+5$ est interprété comme `A+5*ones([n,n])` avec (n,n) la taille de A, et non comme $A+5I_n$.

- Pour les matrices, on peut aussi utiliser la structure `matrix`, dans laquelle le produit entre `matrix` est le produit matriciel. Néanmoins, il est plutôt conseillé d'utiliser la structure `array`.
- Attention aux ensembles de définitions ! Si vous représentez la fonction $f : x \mapsto \frac{1}{x}$ sur $[-10,10]$, selon la valeur du pas ou du nombre de points, on peut calculer $f(0)$.
- Du fait des erreurs d'arrondis, si on fait `arange(xmin,xmax,pas)`, il est possible que le dernier point soit strictement supérieur à `xmax` !

$x \mapsto e^x$	`exp`	exponentiel
$x \mapsto \ln(x)$	`log`	logarithme népérien
$x \mapsto \log(x)$	`log10`	logarithme base 10
$x \mapsto \sqrt{x}$	`sqrt`	racine. Pour la racine n-ième, il faut utiliser `x**(1/n)`
$x \mapsto \cos(x)$	`cos`	cosinus (idem pour sinus et tangente)
$x \mapsto \arccos(x)$	`arccos`	cosinus (idem pour arcsinus et arctangente)
$x \mapsto \lfloor x \rfloor$	`floor`	partie entière mathématique. `ceil` et `round` sont les parties entières par excès et approchée.
$x \mapsto \lvert x \rvert$	`abs`	valeur absolue

⋆ Quelques dessins de courbes représentatives de fonctions

Exercice 3 Dessiner les fonctions suivantes (un dessin par ligne) :

dessin 1 :	$x \mapsto \sin(x)$	$x \mapsto x$	$x \mapsto x - \frac{x^3}{6}$	$x \mapsto x - \frac{x^3}{6} + \frac{x^5}{120}$
dessin 2 :	$x \mapsto e^x$	$x \mapsto 1+x$	$x \mapsto 1+x+\frac{x^2}{2}$	$x \mapsto 1+x+\frac{x^2}{2}+\frac{x^3}{6}$
dessin 3 :	$x \mapsto \ln(1+x)$	$x \mapsto x$	$x \mapsto x - \frac{x^2}{2}$	$x \mapsto x - \frac{x^2}{2} + \frac{x^3}{3}$

Pour chaque dessin, on choisira convenablement l'intervalle d'étude $[$`xmin`$,$`xmax`$]$, ainsi que les axes de manière à montrer que ces polynômes approchent bien ces fonctions au voisinage de 0.

On utilisera des variables pour les valeurs `xmin, xmax, ymin, ymax`. Pensez aussi à afficher la grille avec : `grid()`. Éventuellement, on peut ajouter une légende.

⋆ Autres représentation graphique

Pour dessiner une courbe paramétrée définie par $(x(t),y(t))$ pour $t \in I$, il suffit de discrétiser l'intervalle I en n valeurs $t_1,\dots y_n$, et de créer deux listes (ou deux `1d-array`) `x` et `y` de taille n contenant les valeurs de $[x(t_1),\dots x(t_n)]$ et $[y(t_1),\dots,y(t_n)]$.

La courbe est simplement affichée par la commande `plot(x,y)`.

■ **Exemple V.8** Par exemple, pour la courbe $\begin{cases} x(t) = & \cos(t)(\sin(t)+1) \\ y(t) = & \sin(t)(\cos(t)+1) \end{cases}$ avec $t \in [0,2\pi]$.

On peut faire

```
t = linspace(0, 2*pi, 300) # liste des t: équi-répartis entre 0 et 2%pi.
# liste des (x,y) correspondants:
x = cos(t) * (sin(t)+1)
y = sin(t) * (cos(t)+1)
plot(x,y)
show()
```

■

■ **Exemple V.9** Un autre exemple pour la courbe $\begin{cases} x(t) = & \cos(t)\cos\left(\frac{t}{2}\right) \\ x(t) = & \cos(t)\sin\left(\frac{t}{2}\right) \end{cases}$ avec $t \in [0,4\pi]$.

```
t = linspace(0,4*pi,300)
x = cos(t) * cos(t/2)
y = cos(t) * sin(t/2)
plot(x,y)
```

```
show()
```

■

Exercice 4 Écrire un script qui permette d'afficher les courbes paramétrées suivantes :

$$\begin{cases} x(t) = \cos^3 t \\ y(t) = \sin^3 t \end{cases} \qquad \begin{cases} x(t) = 2\cos t + \cos 2t \\ y(t) = 2\sin t - \sin 2t \end{cases} \qquad \begin{cases} x(t) = \frac{1-t^2}{1+t^2} \\ y(t) = t\frac{1-t^2}{1+t^2} \end{cases} \qquad \begin{cases} x(t) = \frac{t}{1+t^4} \\ y(t) = \frac{t^3}{1+t^4} \end{cases}$$

✯ Correction première partie

```
# -*- coding: utf-8 -*-

"""
Auteur: Sylvain Pelletier
corrections pour la feuille "Possibilités graphiques de la bibliothèque Matplotlib"
"""


# Modules et fonctions importés
################################
from pylab import *

# Fonctions
############
def dessineA():
    plot([1, 2, 3], [1, 3, 1])
    plot([1.5, 2.5], [2, 2])
    show()
    return

def dessineB():
    plot([1, 1, 2, 2, 1.5, 2, 2, 1], [1, 3, 2.7, 2.5, 2.2, 1.8,1.2,1])
    axis([0,3,0,4])
    show()
    return

def DLsinus() :
    print("-"*10+"fonction sinus"+"-"*10)
    xmin = -2*pi
    ymin = -1.2
    xmax = 2*pi
    ymax= 1.2
    x = linspace(xmin,xmax,100)

    plot(x,sin(x), label="sinus")
    plot(x,x, label= "DL 1")
    plot(x,x-(x**3)/6, label= "DL 3")
    plot(x,x-(x**3)/6+(x**5)/120, label= "DL 5")


    axis([xmin, xmax, ymin, ymax])
    plot([xmin,xmax],[0,0], color='black')
    plot([0,0],[ymin,ymax], color='black')

    title("fonction sinus et ses DLs")
    xlabel("x")
    ylabel("y")

    legend()

    show()
```

```
    return
53
def DLexp():
55      print("-"*10+"fonction exponentiel"+"-"*10)
        xmin = -10
57      ymin = -0.2
        xmax = 10
59      ymax= 10
        x = linspace(xmin,xmax,100)
61
        plot(x,exp(x), label= "exponentielle")
63      plot(x,1+x, label = "DL 1")
        plot(x,1+x+(x**2)/2, label = "DL 2")
65      plot(x,1+x+(x**2)/2 + (x**3)/6, label = "DL 3")

67      axis([xmin, xmax, ymin, ymax])
        plot([xmin,xmax],[0,0], color='black')
69      plot([0,0],[ymin,ymax], color='black')

71      title("fonction exponentielle et ses DLs")
        xlabel("x")
73      ylabel("y")

75      legend()

77      show()
        return
79
def DLln():
81      print("-"*10+"x -> ln(1+x)"+"-"*10)
        xmin = -0.1
83      ymin = -20
        xmax = 5
85      ymax= 10
        x = linspace(10**(-10),xmax,100)
87
        plot(x,1+log(x), label="ln")
89      plot(x,x, label= "DL 1" )
        plot(x,x-(x**2)/2, label = "DL 2")
91      plot(x,x-(x**2)/2 + (x**3)/3, label = "DL 3")

93      axis([xmin, xmax, ymin, ymax])
        plot([xmin,xmax],[0,0], color='black')
95      plot([0,0],[ymin,ymax], color='black')

97      title("fonction logarithme et ses DLs")
        xlabel("x")
99      ylabel("y")

101     legend()

103     show()
        return
105


107 def courbeParam1() :
        t = linspace(0,2*pi,200)
109     x = (cos(t))**3
        y = (sin(t))**3
111     plot(x,y)
        show()
113
def courbeParam2() :
115     t = linspace(0,2*pi,200)
```

```
    x = 2*cos(t) + cos(2*t)
    y = 2*sin(t) - sin(2*t)
    plot(x,y)
    show()

def courbeParam3() :
    t = linspace(-10,10,200)
    x = (1-t**2)/(1+t**2)
    y = t* (1-t**2)/(1+t**2)
    plot(x,y)
    show()

def courbeParam4() :
    t = linspace(-10,10,200)
    x = t/(1+t**4)
    y = t**3/(1+t**4)
    plot(x,y)
    show()



# Code:
#######
dessineA()
dessineB()
DLsinus()
DLexp()
DLln()

courbeParam1()
courbeParam2()
courbeParam3()
courbeParam4()
```

Même si cela est moins utile, il peut être intéressant de représenter des informations à deux dimensions, c'est le cas en particulier pour dessiner une fonction $f : \mathbb{R}^2 \to \mathbb{R}$.

Pour cela, il faut calculer une matrice `M` dont l'élément `(i,j)` contiendra les valeurs de $f(x_i, y_j)$, pour des points (x_i, y_j).

La commande plus simple est `matshow(M)` qui dessine le contenu de la matrice `M` sous forme de couleur.

■ **Exemple V.10** Pour montrer les 12 premières couleurs

```
M= [ [i+j for i in range(6)] for j in range(6)]
matshow(M)
show()
```

■

Enfin, la commande `bar` permet de tracer des histogrammes de valeurs. La syntaxe conseillée est

```
bar(classes , effectifs, align = 'center')
```

où `classes` est la liste des classes, et `effectifs` la liste des effectifs de chaque classe.

On peut aussi faire tout simplement `bar(effectifs)`.

Cette technique est particulièrement utile pour déterminer la loi empirique d'une variable aléatoire réelle.

■ **Exemple V.11** Pour illustrer :

```
effectifs = [ 13, 15 , 12 ,10 ]
classes = [1,2,3,4]
bar(classes, effectifs, align = 'center')
show()
```

■

Exercice 5

1. Écrire une fonction qui renvoie la valeur de la somme de deux dés.
   Indication : la fonction `rand()` renvoie un nombre aléatoire suivant la loi uniforme sur $]0,1[$. Que renvie la commande : `int( 6* rand()) +1` ?
2. Faire 1000 tests en construisant une liste `effectifs` de taille 11 tel que : pour $i \in [\![0,10]\!]$, la valeur `effectifs[i]` est le nombre de résultats égaux à $i+2$.
3. Construire l'histogramme des résultats : on doit observer un pic à 7 et une décroissance de chaque côté de ce pic.

Correction :

```
"""
loi empirique de la somme de deux dés
"""
from pylab import *

def simule():
    de1 = int(6*rand()+1)
    de2 = int(6*rand()+1)
    return de1 + de2

effectifs = [0] * 11
for test in range(1000):
    resultat = simule()
    effectifs[ resultat-2] += 1
bar(range(2,13), effectifs, align = 'center')
show()
```

Pour faire des petites animations :

- on efface le graphique avec `clf()`
- on dessine le graphique, dans une échelle fixée !
- on montre avec l'instruction `show(block = False)`, ie sans obliger à fermer la fenêtre.
- on utilise la commande `pause(0.1)` ; qui permet d'attendre (le 0.1 est en secondes, on peut régler cette valeur).
- et on recommence.

R Cette méthode permet simplement d'obtenir des résultats qualitatifs sur l'animation : selon la charge du processeur l'animation peut ralentir.

On ne peut donc pas y effectuer de mesures.

Il existe une meilleur méthode pour faire des animations, mais cela dépasse largement le programme.

Exercice 6 Cordes vibrantes

On considère une corde de guitare fixée à chacune de ses extrémités. Lorsqu'on joue une note, la corde se met à vibrer. Nous allons modéliser numériquement son mouvement.

La corde à l'instant t est représentée par le graphe d'une fonction $x \mapsto u(t,x)$, pour $x \in [0,l]$. La fonction $u(t,x)$ représente la déformation verticale de la corde à l'instant $t \geqslant 0$ et à la position $x \in [0,l]$.

Au repos, la corde est tendue, et donc représentée par un segment de longueur l, *i.e.* la fonction $u(0,.)$ est nulle sur $[0,l]$.

La physique nous apprend que u s'écrit comme une combinaison linéaire des mouvements fondamentaux appelés *modes* donnés par :

$$(t,x) \mapsto A_n \cos(nwt + \phi_n) \sin(nwx/l)$$

pour tout entier n non nul (w est la pulsation de la corde, déterminée par ses caractéristiques physiques, et ϕ est le déphasage).

On choisit dans la suite $w = 1$ et $l = 1$.

1. **Mouvement principal**

Le mouvement principal correspond au premier mode. Dans ce cas, on a :

$$\forall (t,x) \in \mathbb{R}_+ \times [0,l], \qquad u(t,x) = A_1 \cos(t) \sin(\pi x)$$

A_1 s'appelle *l'harmonique principale*.
En prenant $A_1 = 1$ représenter la vibration de la corde pour t entre 0 et 20.
Pour cela :

- On fera une boucle sur t : `for t in arange(0,20,0.05)`, la valeur `0.05` corresponds alors au pas de temps.
- Pour chacun de ces `t`, on effacera le graphique et on représentera la fonction : $y = \cos(t)\sin(\pi x)$, en fixant le cadre aux valeurs `axis([0,1,-2,2])`.
- Choisir la valeur de la pause adéquate.

2. **Mouvement secondaire**
Reprendre la question 1 en superposant les deux premiers modes, c'est-à-dire le cas où :

$$\forall (t,x) \in \mathbb{R}_+ \times [0,l], \qquad u(t,x) = A_1 \cos(t) \sin(\pi x) + A_2 \cos(2t + \phi_2) \sin(2\pi x).$$

On prendra $A_2 = 0.1$ et ϕ_2 aléatoire entre $[0, 2\pi]$ (la phase est bien sûr fixée une fois pour toute).

Correction :

```
"""
animation graphique des cordes vibrantes
"""
from pylab import *

## mouvement principal
x = linspace(0,1,200)
for t in arange(0,10, 0.05) :
    y = cos(t) * sin(pi*x)
    clf()
    plot(x,y)
    axis([0,1,-2,2])
    show(block = False)
    pause(0.0001)

## mouvement secondaire
phi2 = rand()*2*pi # phase aléatoire
for t in arange(0,10, 0.05) :
    y = cos(t) * sin(pi*x) + 0.1 * cos(2*t+phi2) * sin(2*pi*x)
    clf()
    plot(x,y)
    axis([0,1,-2,2])
    show(block = False)
    pause(0.0001)
```

⋆ Dessins de fractales

Exercice 7 **Courbes de Hilbert**

Soit une matrice A de la forme :

$$A = \begin{bmatrix} x_0 & y_0 \\ \vdots & \vdots \\ x_{n-1} & y_{n-1} \end{bmatrix}.$$

Cette matrice correspond à une courbe $\mathscr{C}$, reliant les points $(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})$ par des segments.

1. Quelle instruction permet de tracer la courbe $\mathscr{C}$ correspondant à la matrice A en utilisant comme cadre le carré $[0,1]^2$?

La *transformée de Hilbert* consiste à construire une nouvelle matrice $B = \phi(A)$, telle que :

$$B = \begin{bmatrix} X_0 & Y_0 \\ \vdots & \vdots \\ X_{4n-1} & Y_{4n-1} \end{bmatrix}.$$

Telle que pour $i = 0 \ldots n-1$

- $(X_i, Y_i) = \frac{1}{2}(y_i, x_i)$,
- $(X_{n+i}, Y_{n+i}) = \frac{1}{2}(x_i, 1+y_i)$,
- $(X_{2n+i}, Y_{2n+i}) = \frac{1}{2}(1+x_i, 1+y_i)$,
- $(X_{3n+i}, Y_{3n+i}) = \frac{1}{2}(2-y_i, 1-x_i)$.

2. Écrire une fonction `phi`, qui prend en entrée la matrice A et retourne la matrice B.
Tester sur la matrice :

```
 A  =
    0.25    0.25
    0.25    0.75
    0.75    0.75
    0.75    0.25
```

pour laquelle

```
-->phi(A)
 ans  =
    0.125    0.125
    0.375    0.125
    0.375    0.375
    0.125    0.375
    0.125    0.625
    0.125    0.875
    0.375    0.875
    0.375    0.625
    0.625    0.625
    0.625    0.875
    0.875    0.875
    0.875    0.625
    0.875    0.375
    0.625    0.375
    0.625    0.125
    0.875    0.125
```

On s'intéresse aux itérés de la matrice A, c'est-à-dire à la suite de matrices : $A^0 = A$, $A^1 = \phi(A)$, $A^2 = \phi(A^1) = \phi \circ \phi(A)$, etc.

3. Écrire un script qui initialise A, comme ci-dessus, et trace la courbe correspondante, puis trace la courbe correspondante à A^i pour i variant entre 1 et `niter`.
Pour chaque A^i, on affichera le graphique.
N.B. : On s'arrêtera à `niter=5`, sinon les calculs sont trop longs.
4. Changer la valeur de A (tout en gardant des valeurs entre 0 et 1), et relancer le script.
On pourra par exemple prendre des valeurs aléatoires.

Correction :

```
"""
courbes fractales de Hilbert
"""
from pylab import *

def phi(A):
    """
    entrée A = 2d array à 2 colonnes
             = coordonnes des n points de la courbe à l'instant t
    sortie B = 2d array à 2 colonnes
             = coordonnes des 4n points de la courbe à l'instant t+1
    """
    n,m = shape(A)
    B = zeros((4*n,2))
    for i in range(n):
```

```
        B[i, :] = 0.5* array( [A[i,1] , A[i,0] ] )
        B[n+i, :] = 0.5*array([ A[i,0], 1+ A[i,1]])
        B[2*n+i, :] = 0.5*array([1+A[i,0], 1+ A[i,1]])
        B[3*n+i, :] = 0.5*array([2-A[i,1], 1- A[i,0]])
    return B


A = array( [[0.25,0.25], [0.25,0.75], [0.75, 0.75], [0.75, 0.25]])
# A =rand( 4,2)
for i in range(5):
    plot(A[:,0], A[:,1])
    axis([0,1,0,1])
    show()
    A=phi(A)
```

Exercice 8 **Ensemble de Mandelbrot**

Soit un nombre complexe $c \in \mathbb{C}$, on construit la suite récurrente $\left(u_n^c\right)_{n\in\mathbb{N}}$ en fonction du paramètre c par :

$$u_0^c = 0, \qquad \text{et} \qquad \forall n \in \mathbb{N},\ u_{n+1}^c = (u_n^c)^2 + c.$$

L'ensemble de Mandelbrot est l'ensemble des points c tels que la suite $\left(u_n^c\right)$ ne tend pas vers l'infini en module :

$$\overline{\mathscr{M}} = \left\{ c \in \mathbb{C} \middle| \lim_{n\to+\infty} |u_n^c| = +\infty \right\}$$

On souhaite représenter cet ensemble, c'est-à-dire que l'on souhaite afficher une image, où un point de coordonnées (x,y) est :

- noir si et seulement si la suite des itérés (u_n^c) construite en utilisant le paramètre $c = x+iy$ ne tend pas vers $+\infty$ en module, ie $x+iy \in \mathscr{M}$
- blanc sinon.

Il peut être prouvé que si pour un certain entier $n_0 \in \mathbb{N}$, on a $|u_{n_0}^c| > 2$, alors la suite tend vers $+\infty$ en module et donc $c \notin \mathscr{M}$.

Pour déterminer si un complexe $c \in \mathscr{M}$, on calcule donc la suite des premiers itérés. Si au bout d'un certain nombre d'itération on a toujours $|u_n^c| \leqslant 2$, alors on considère que la suite ne tends pas vers $+\infty$ (en module). Si au cours des itérations on a une valeur n_0 telle que $|u_{n_0}^c| > 2$, alors on considère que le point $c \notin \mathscr{M}$.

1. Écrire une fonction `suiteIter` qui prend en entrée : un entier `niterMax` et un complexe `c`. Cette fonction construit les `niterMax` premiers itérées et renvoie la variable `diverge`.
   Cette variable booléenne contiendra `True` si au cours des `niterMax` premières itérations, le module de `z` dépasse 2 (on arrêtera le calcul dans ce cas), `False` sinon.
   Rappel : le type complexe s'utilise facilement en Python : le nombre complexe $x+iy$ s'écrit : `x+y*1J`. de plus le module s'obtient avec la fonction `abs`.
2. Écrire un script qui :
   - crée deux array de taille `m` : `x` et `y` contenant des valeurs équi-réparties dans $[-2,0.5]$ et $[-1.25,1.25]$ respectivement.
     On a ainsi discrétisé le rectangle $[-2,0.5]\times[-1.25,1.25]$ en m^2 points.
   - Pour toutes les valeurs de `i,j` entre 0 et `m-1`, utilise la fonction `suiteIter` pour déterminer si le complexe `c` défini par `c=x[i]+y[j]*1J` est dans l'ensemble de Mandelbrot.
     Précisément, on créera un `2d-array` `img` telle que pour chaque couple `(i,j)`, `img[i,j]=255` si la fonction la valeur de `diverge` en partant de `c` est `False` et `img(i,j)=0` sinon.
   - Pour afficher la matrice `img`, on utilisera `matshow(img)`.
3. Pour faire un zoom, changer l'intervalle de valeurs de `x` et `y`. Par exemple $[-0.4,0.2]\times[0.6,1]$.

On pourra prendre pour valeurs raisonnables : `niterMax=30` et `m=50` (attention, les calculs deviennent vite très longs).

Correction :

```
"""
fractales de Mandelbrot
"""
```

```
4
## import
6 from pylab import *

## fonctions
8 def suiteIter(niterMax, c):
    """
10     entrée: niterMax = int
                = maximum des itérations
12              c = complexe
                = paramètre de la suite
14     sortie: diverge = bool
                = False si au bout de niterMax itérations, la suite est toujours de module
16     inférieur à 2 true sinon

18
    """
20     u = 0
    i = 0
22     diverge = False
    while not(diverge) and i <= niterMax :
24         u=u**2 + c
        i += 1
26         if abs(u)>2 :
            diverge = True
28
    return diverge
30
## code
32 m = 200
img = zeros( (m,m))
34 x = linspace(-2, 0.5, m)
y = linspace(-1.5, 1.25, m)
36 for i in range(m):
    for j in range(m):
38         c = x[i] + y[j]*1J
        if niter( 100, c) :
40             img[i,j] = 255
        else:
42             img[i,j] = 0
matshow(img)
44 show()
```

R L'ensemble de Mandelbrot bien que venant d'un problème en apparence très simple est en fait très complexe : il s'agit d'un **ensemble fractal** qui a des détails similaires à des échelles arbitrairement petites ou grandes.

Pour en savoir plus sur l'ensemble de Mandelbrot :
http://fr.wikipedia.org/wiki/Ensemble_de_Mandelbrot

Exercice 9 **Carpette de Sierpinsky**

Soit XYZ le triangle équilatéral du plan, dont les coordonnées des sommets sont :

$$X = (0,0), \qquad Y = (2,0), \qquad Z = (1, \frac{\sqrt{3}}{2})$$

On construit une suite aléatoire de points $(M_n)_{n\in\mathbb{N}}$ telle que, pour tout $n \geqslant 0$, M_{n+1} est le milieu du segment reliant M_n à l'un des trois points X, Y ou Z, choisi au hasard.

Dans cet exercice, on choisit de représenter un point par un couple de flottants (les coordonnées), plutôt que par un `1d-array`.

1. Écrire une fonction `milieu(A,B)` qui prend en entrée deux listes `A` et `B` contenant les coordonnées (x_A, y_A) du point `A` (resp. du point `B`) et calcule une liste `I`, qui contient les coordonnées du milieu du segment $[A,B]$.
2. Écrire une fonction `pointsuivant(A)`, qui étant donné un vecteur `A` qui contient les coordonnées du point M_n,

calcule les coordonnées du point suivant (en appelant la fonction `milieu`). On choisira au hasard entre les points X, Y ou Z.
Indication : on utilisera la fonction `rand()` et on regardera si le nombre aléatoire obtenu est entre dans $\left[0, \frac{1}{3}\right]$ ou dans $\left[\frac{1}{3}, \frac{2}{3}\right]$ ou dans $\left[\frac{2}{3}, 1\right]$.

3. Écrire un script `carpette` qui utilise ces fonctions. On commencera par `A=[0,0]`, puis itérera `nPoints` fois la fonction `pointsuivant`.
   À chaque itération, on ajoutera au graphique le point `A`, par la commande : `plot(A[0], A[1], marker=',', color='black'`
   On montrera le graphique après la boucle `for` avec `show()`

Pour des valeurs raisonnables, on commencera avec `nPoints=1000`

Correction :

```
"""
fractales carpette de Sierpinsky
"""

from pylab import *

## fonctions
def milieu(A,B):
    """
    entrée: A et B = couples de float
            = coordonnées points A et B
    sortie C = couple de float
             = coordonées du milieu C de [A,B]
    """
    Cx = 0.5 *(A[0]+B[0])
    Cy = 0.5 *(A[1]+B[1])
    return [Cx, Cy]

def pointsuivant(A):
    alea =rand()
    if alea < 0.333 :
        return milieu(A, [0,0])
    elif alea < 0.666 :
        return milieu(A, [2,0])
    else:
        return milieu(A, [ 1, sqrt(3)/2])

nPoints = 10000
A = [0,0]
for i in range(nPoints):
    plot(A[0], A[1], marker=',', color='black')
    axis([0,2,0,2])
    A = pointsuivant(A)
axis([0,2,0,1])
show()
```

Jeux de Nim en une dimension

Pelletier Sylvain PSI, LMSC

Les jeux de Nim sont des jeux de stratégie pure, à deux joueurs. Il en existe plusieurs variantes.

Dans ce td, on va explorer la version la plus simple pour illustrer les techniques d'algorithmique pour l'intelligence artificielle et l'étude des jeux. Le prochain td sera consacré à une version un peu plus compliquée.

Dans cette première version donc, les deux joueurs disposent au départ d'un tas de N allumettes. Ils voient le tas (et connaissent donc la valeur de N). Chaque joueur à son tour retire 1, 2 ou 3 allumettes et le perdant est celui qui prend la dernière.

On cherche à étudier ce jeu pour savoir si le premier joueur peut gagner à coup sûr (selon la valeur de N), déterminer la stratégie dans ce cas et écrire une intelligence artificielle qui soit capable de jouer le meilleur coup.

Dans cette version, il s'agit bien évidemment d'un « problème jouet » et en y réfléchissant un peu, vous trouverez facilement la solution : la stratégie est de laisser un nombre d'allumettes congru à 1 modulo 4 à l'adversaire.

Dans la position de départ :

- si $N \equiv 1[4]$, alors le joueur 1 a perdu : si il enlève k allumette alors le joueur 2 enlève $4-k$ allumettes pour lui laisser toujours un nombre d'allumettes congru à 1 modulo 4.
- si $N \equiv 0[4]$, alors le joueur 1 prend 3 allumettes et donc donne au joueur suivant une valeur de N congru à 1 modulo 4.
- de même si $N \equiv 2[4]$, alors le joueur 1 prend 1 allumette, si $N \equiv 3[4]$, alors le joueur 1 prend 2 allumettes.

Le but de ce td est de retrouver ce résultat en utilisant un algorithme pour le généraliser à des cas plus complexes. On s'interdit donc d'utiliser la solution ci-dessus.

On introduit donc le vocabulaire suivant :

On appelle état du jeu la valeur de N que reçoit le joueur qui doit jouer.

Cela correspond donc à un sommet dans un graphe orienté non pondéré. Un état N_1 est relié à un état N_2 si il existe un coup du joueur qui permet de passer de l'état N_1 à l'état N_2.

Dans notre exemple donc l'état N_1 est relié aux états N_1-1, N_1-2 et N_1-3.

On va classifier les états en deux parties disjointes :

- Un état est perdant en entrée (ou simplement perdant) si le joueur qui reçoit cet état perd la partie quels que soient ses choix si son adversaire joue parfaitement,
- Un état est gagnant en entrée (ou simplement gagnant) si le joueur qui reçoit cet état peut choisir un coup qui lui assure de gagner la partie quelle que soit la manière dont joue son adversaire.

Ainsi un état est gagnant en entrée si il est relié dans le graphe à un état perdant (ie si on peut trouver un coup qui laisse l'adversaire dans un état perdant) et un état est perdant en entrée si il n'est relié qu'à des états gagnants.

Le but de l'étude d'un jeu de Nim est donc de classifier les états et d'être capable pour un état gagnant de donner le meilleur coup.

Dans notre exemple :

- l'état $N=1$ est perdant de manière évidente.
- les états $N=2,3$ et 4 sont donc gagnants, car reliés à un état perdant. Précisément, pour l'état 2, le coup à jouer est 1 (ie enlever une allumette), pour l'état 3, le coup à jouer est 2, pour l'état 4 le coup à jouer est 3.
- l'état 5 est perdant, car il n'est relié qu'à des états gagnants.
- En conséquence, les voisins de 5 sont gagnants.

Le but de ce td est donc de créer :

- la liste `etatPerdant` contenant toutes les valeurs des états perdants
- le dictionnaire `etatGagnant` de la forme `i:j`, où i est un état gagnant et j le nombre d'allumettes à enlever pour gagner.

Pour cela, on part de l'état 1 et on balaie tout le graphe, en étudiant les états dans l'ordre : il est facile de classifier l'état i, en ayant classifié tous les états précédents.

Par exemple, pour $N=15$, on obtiendra le résultat suivant :

```
etatPerdant = [1, 5, 9, 13]
etatGagnant = {2: 1, 3: 2, 4: 3, 6: 1, 7: 2, 8: 3, 10: 1, 11: 2, 12: 3, 14: 1, 15: 2}
```

Correction : on peut par exemple faire :

```
"""
TD jeu de nim en 1D.

crée la liste etatPerdant et le dictionnaire etatGagnant
"""


def IACreeListe(N):
    """
    entrée: N = nbr d'alummettes dans le jeu
    sortie: etatPerdant
    = list de int
    = liste des états perdants
    etatGagnant = dictionnaire de la forme i : j
    i et j int
    = l'état i est gagnant et il faut enlever j allumettes
    """

    # initialisation:
    etatPerdant = [1]
    etatGagnant = {}

    for i in range(2, N+1):
        # on regarde si i-1 ou i-2 ou i-3 est perdant
        # si c'est le cas, on enlève le nombre d'alummettes
        # pour arriver à un état perdant
        if i-1 in etatPerdant :
            etatGagnant[i] = 1
        elif i-2 in etatPerdant :
            etatGagnant[i] = 2
        elif i-3 in etatPerdant :
            etatGagnant[i] = 3
        else :
            # aucun voisin n'est perdant
            # donc tous les voisins sont gagnants
            # l'état i est donc perdant
            etatPerdant.append(i)
    return etatGagnant, etatPerdant


N = 15
etatGagnant, etatPerdant = IACreeListe(N)
print("pour la valeur de départ N= ", N)
print("on obtient les listes")
print("liste des états perdants: ", etatPerdant)
print("dictionnaire des états gagnants: ", etatGagnant)
```

Jeux de Nim (version améliorée)

Pelletier Sylvain PSI, LMSC

Dans ce TD, on va reprendre les principes vus pour le jeu de Nim en une dimension, mais on va les appliquer à une version plus compliquée pour laquel la résolution analytique n'est pas possible : les deux joueurs disposent d'allumettes disposées selon k lignes : la première ligne contient 1 allumette, la deuxième 2 allumettes, la troisième 3, etc. Le nombre total d'allumettes au départ est donc $N = k(k+1)/2$. À chaque tour, le joueur choisit une ligne et enlève dans cette ligne 1, 2 ou 3 allumettes. Le perdant étant celui qui prend la dernière.

Bien que le problème soit plus complexe, la méthode d'analyse du jeu est la même que dans notre « problème -jouet » :

- l'état du jeu est représenté par une liste d'entiers (positifs ou nuls) de longueur k : il s'agit du nombre d'allumettes sur chaque ligne.
  L'état de départ est donc : $[1,2,3,4,5,\ldots,k]$, ce qui correspond à (pour $k=4$) :

```
*
**
***
****
```

  L'état $[1,0,2,4]$ correspond donc à

```
*

**
****
```

  Pour un état e, on pourra noter le nombre total d'allumettes noté $N(e)$.
- Chaque état correspond à un sommet dans un graphe orienté et non pondéré. Un état e_1 est relié à l'état e_2 si le joueur qui reçoit l'état e_1 peut transmettre l'état e_2.
  Notons en particulier que dans ce cas : $N(e_2)$ est égal à $N(e_1)-1$ ou $N(e_1)-2$ ou $N(e_1)-3$.
  On peut coder le chemin de e_1 vers e_2, ie le coup que peut jouer le joueur pour passer de e_1 à e_2 sous la forme d'un couple : (l,a), où $l \in [\![0,k-1]\!]$ est un numéro de ligne et $a \in [\![1,3]\!]$ le nombre d'allumettes à enlever.
- Le but de l'étude de ce jeu consiste donc à partitionner ce graphe en deux parties : les états perdants et les états gagnants.
  De la même manière que précédemment :
  - un état est perdant si il n'est relié qu'à des états gagnants.
  - un état est gagnant si il est relié à au moins un état perdant.

  Il faut de plus noter pour chaque état gagnant quel coup (i,j) jouer pour aller à un état perdant.

Le but de ce td est donc de nouveau de créer :

- la liste `etatPerdant` contenant toutes les valeurs des états perdants, il s'agira d'une liste d'états, un état étant une liste de longueur k de int.
- le dictionnaire `etatGagnant` de la forme `etat: coup`, où `etat` est un état gagnant (représenté sous forme d'une liste donc) et `coup` est le coup à jouer pour gagner (représenté sous la forme d'un couple (l,a), numéro de ligne et nombre d'allumettes à enlever).

On peut imaginer plusieurs méthodes pour cela, toutes vont procéder en parcourant le graphe par valeur de N croissante :

- Les états e tels que $N(e)=1$ sont perdants.
- On classifie l'état e tel que $N(e)=n$ lorsque l'on a traité tous les états avec une valeurs de N strictement inférieure à n. Il faut donc générer tous les sommets du graphe (ie les états) en les classant par ordre croissant de valeur de N.
- On commence donc par les états tels que $N(e)=1$.
  Puis pour chaque état (dans l'ordre), on regarde comment sont classés tous ses voisins (qui sont tous classifiés). Si l'un des états voisins est perdant, alors l'état est gagnant. Si tous les états voisins sont gagnants, alors l'état est perdant.

Voici le découpage proposé du programme :

- On va tout d'abord générer tous les états par un algorithme récursif, sans les trier. Précisément, on va aussi générer l'état `[0, .... ,0]` (ie pas d'allumette) pour que ce soit plus simple. On va donc écrire une fonction :

```
def listeEtatEtendu(k):
    """
```

```
    entrée: k = int = nbr de lignes
    sortie: lEtat = liste des etats etendus (avec [0,....,0])
            de taille k non triée
    """
```

Pour cela, il y a deux états pour $k = 1$ (aucune allumette ou une seule). Puis pour un entier k, il faut générer tous les états à $k-1$ lignes et ajouter à chacun les valeurs $0, 1, \ldots, k$.
(**REM :** un exercice élémentaire de dénombrement indique qu'il y a $(2 \times 3 \times \ldots k+1) = (k+1)!$ états étendus. Bien vérifier cette fonction avant de passer à la suite.

- Ensuite, on va trier cette liste. Pour cela, on va appliquer le cours en utilisant un algorithme de tri.
  Il faut tout d'abord être capable de comparer deux états pour savoir si l'un est avant l'autre. On va donc écrire une fonction :

```
def estAvant(etat1, etat2):
    """
    entrée: etat1 = liste de int = etat du jeu
    etat2 idem
    sortie: Booleen = True si etat1 contient moins d'allumettes que etat2
    False sinon.
    """
```

Cette fonction compare tout simplement le nombre d'allumettes dans les deux états.
Ensuite, on appliquera (par exemple) le tri par insertion, en écrivant une fonction :

```
def triInsertion(L):
    """
    entrée: L = list d'éléments quelconques
    comparable par la fonction estAvant
    sortie: L triée
    """
```

Pour vérification, on a (l'ordre peut éventuellement être différent selon votre algorithme de tri) pour $k = 3$:

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

- Maintenant on va parcourir le graphe. Il faut donc être capable étant donné un état de construire la liste des états voisins.
  Écrire une fonction `voisins(etat)` qui étant donné un état construit la liste des voisins de cet état dans le graphe, ie la liste des états que l'on peut obtenir à partir de `etat`.
  Cette fonction sortira une liste de couples de la forme `[etatVoisin, coup]` où `etatVoisin` est donc un des états voisin de `etat` et `coup` est le coup à jouer pour les relier.

```
def voisins(etat):
  """
  entrée: etat = liste de int = etat du jeu
  sortie: lVoisinCoup = liste de couples de la forme [etatVoisin, coup]
          avec etatVoisin un état (list de int)
          et coup le coup à jouer pour passer de etat etatVoisin
          (couple de int (l,a) avec
                          l = num de ligne et
                          a = nbr d'allumettes à enlever)
  """
```

Pour vérifier, on pourra utiliser les instructions :

```
for etat in lEtat :
    lvc = voisins(etat)
    for voi, coup in lvc:
        print("en partant de ", etat, "et en jouant ", coup, " on obtient", voi)
```

qui devrait afficher des lignes du type :

```
en partant de  [0, 2, 0] et en jouant  [1, 2]  on obtient [0, 0, 0]
en partant de  [1, 2, 3] et en jouant  [1, 2]  on obtient [1, 0, 3]
```

- Il ne reste plus qu'à classifier les états. Pour cela, on va écrire une fonction

```
def classifie(k):
    """
    entrée: k = int = nbr de lignes
    sortie: etatPerdant = list de etat (list de int)
                        = liste des états perdants
            etatGagnant = list de couple [etat:coup]
                    où etat  est un état gagnant
                    et coup est un coup à jouer.
    """
```

qui procédera comme indiqué ci-dessus. Par convention, l'état sans allumette sera gagnant avec le coup $[0,0]$.
Détail technique : comme Python ne peut pas « hacher » les listes, il faut que les clés d'un dictionnaire soient des tuples et non des listes. On écrire donc l'instruction :

```
etatGagnant[tuple(etat)] = coup
```

pour ajouter une clé dans le dictionnaire `etatGagnant` égale à `coups`.

Pour vérifier, on pourra obtenir :

```
# états gagnants:
{(0, 0, 0): [0, 0], (0, 0, 2): [2, 1], (0, 1, 1): [1, 1],
(0, 2, 0): [1, 1], (1, 0, 1): [0, 1], (1, 1, 0): [0, 1],
(0, 0, 3): [2, 2], (0, 1, 2): [2, 2], (0, 2, 1): [1, 2],
(1, 0, 2): [2, 2], (1, 2, 0): [1, 2], (0, 1, 3): [2, 3],
(1, 0, 3): [2, 3], (1, 1, 2): [2, 1], (1, 2, 1): [1, 1],
(0, 2, 3): [2, 1], (1, 1, 3): [2, 2], (1, 2, 2): [0, 1]}
#états perdants:
[[0, 0, 1], [0, 1, 0], [1, 0, 0], [1, 1, 1], [0, 2, 2], [1, 2, 3]]
```

Correction :

```
from pylab import rand

def listeEtatEtendu(k):
    """
    entrée: k = int = nbr de lignes
    sortie: lEtat = liste des etats etendus (avec [0,....,0])
            de taille k non triée
    """
    if k == 1 :
        return [[0], [1]]
    lEtat = []
    lPrec = listeEtatEtendu(k-1)
    for etat in lPrec :
        for i in range(k+1):
            tmp = etat.copy()
            tmp.append(i)
            lEtat.append(tmp)
    return lEtat

def estAvant(etat1, etat2):
    """
    entrée: etat1 = liste de int = etat du jeu
    etat2 idem
    sortie: Booleen = True si etat1 contient moins d'allumettes que etat2
    False sinon.
    """
    S1 = sum(etat1)
    S2 = sum(etat2)
    if S1 < S2:
        return True
    return False

def triInsertion(L):
```

```
    """
    entrée: L = list d'éléments quelconque
    comparable par la fonction estAvant
    sortie: rien, la liste L est triée
    """
    n= len(L)
    for k in range(1, n):
        i = k
        while i>0 and estAvant(L[i], L[i-1]):
            L[i], L[i-1] = L[i-1], L[i]
            i  -= 1



def voisins(etat):
    """
    entrée: etat = liste de int = etat du jeu
    sortie: lVoisinCoup = liste de couple de la forme [etatVoisin, coup]
            avec etatVoisin un état (list de int)
            et coup le coup à jouer pour passer de etat etatVoisin
            (couple de int (l,k) avec
                            l = num de ligne et
                            k = nbr d'allumettes à enlever)
    """
    k = len(etat) # nbr de lignes
    lVoisinCoup = []
    for l in range(k):
        for a in range(1, 4):
            if etat[l]>= a :
                tmp = etat.copy()
                tmp[l] -= a
                lVoisinCoup.append( [tmp, [l,a]])
    return lVoisinCoup


def classifie(k):
    """
    entrée: k = int = nbr de lignes
    sortie: etatPerdant = list de etat (list de int)
                        = liste des états perdants
           etatGagnant = list de couple [etat:coup]
                    où etat  est un état gagnant
                    et coup et un coup à jouer.
    """
    lEtat = listeEtatEtendu(k)
    triInsertion(lEtat)

    etatPerdant  = []
    etatGagnant  = {}

    for etat in lEtat :
        if etat == [0]*k :
            etatGagnant[tuple(etat)] = [0,0]
        elif sum(etat) == 1 :
            etatPerdant.append(etat)
        else :
            lvc = voisins(etat)
            estGagnant = False
            for voisin, coup in lvc:
                if voisin in etatPerdant :
                    etatGagnant[tuple(etat)] = coup
                    estGagnant = True
                    break
            if not estGagnant :
```

```
                etatPerdant.append(etat)
    return etatGagnant, etatPerdant

def coupAlea(etat):
    """
    entrée: etat = liste de int = etat du jeu
    sortie: coup=
            (couple de int (l,k) avec
                            l = num de ligne et
                            k = nbr d'allumettes à enlever)
            coup aléatoire dans la liste des coups possibles
    """
    lcv = voisins(etat)
    n = len(lcv)
    alea = int(rand()*n)
    etatAlea, coupAlea = lcv[alea]
    return coupAlea

def coupIA(etat, etatGagnant):
    """
    entrée: etat = liste de int = etat du jeu
            etatGagnant = list de couple [etat:coup]
                où etat  est un état gagnant
                et coup et un coup à jouer.
    sortie: coup=
            (couple de int (l,k) avec
                            l = num de ligne et
                            k = nbr d'allumettes à enlever)
            coup gagnant si gagnant sinon aléatoire
    """
    if tuple(etat) in etatGagnant :
        print("tu as déjà perdu mais tu ne le sais pas encore")
        return etatGagnant[tuple(etat)]
    else :
        print("je tente ma chance")
        return coupAlea(etat)


def coupHuman(etat):
    """
    entrée: etat = liste de int = etat du jeu
    sortie: coup=
            (couple de int (l,k) avec
                            l = num de ligne et
                            k = nbr d'allumettes à enlever)
            demande le coup au joueur
    """
    nbrLigne = len(etat)
    ok = False
    while  not ok:
        ok = True
        try :
            l = int(input("numéro de ligne (en commençant à 0)"))
            assert 0<=l < nbrLigne
            k = int(input("Combien d'allumettes"))
            assert k <= etat[l]
        except :
            ok = False
    return (l,k)

def affiche(etat):
    """
    entrée: etat = liste de int = etat du jeu
    sortie: rien affichage
```

```
    """
    for l in range(len(etat)) :
        print("|"+"*"*etat[l])


def playAGame(k):
    """
    entrée: k = int = nbr de lignes
    sortie: rien
    """

    lg , lp = classifie(k)
    etat = [i+1 for i in range(k)]
    tour = 0

    # mettre dans la liste game
    # deux valeurs parmis
    # aleatoire, humain, IA
    game = ["humain", "IA"]




    print("début du jeu entre", game[0], " et ", game[1])
    while sum(etat ) >0:
        affiche(etat)
        print(game[tour%2], " joue ")
        if game[tour%2] == "IA" :
            (ligne, nbrAll) = coupIA(etat, lg)
        elif game[tour%2] == "aleatoire" :
            (ligne, nbrAll) = coupAlea(etat)
        elif game[tour%2] == "humain" :
            (ligne, nbrAll) = coupHuman(etat)
        print("coup choisi: enlever", nbrAll," allumettes sur la ligne ",  ligne)
        etat[ligne] -= nbrAll
        tour += 1
    print("fin du jeu, victoire de ", game[tour%2])




print("test du programme nim vers 2d")
k = 3
print("avec ", k, "lignes")

"""
test pour générer tous les états et les trier
"""
lEtat = listeEtatEtendu(k)
print("listes des états étendus avant tri:")
print(lEtat)
print("il y en a", len(lEtat))
print("listes des états étendus après tri:")
triInsertion(lEtat)
print(lEtat)


"""
vérification de la liste des voisins
"""
for etat in lEtat :
    lvc = voisins(etat)
    for voisin, coup in lvc:
        print("en partant de ", etat, "et en jouant ", coup, " on obtient", voisin)
```

```
227 """
    classification
229 """
    lg , lp = classifie(k)
231 print("liste des états gagnants")
    print(lg)
233 print("liste des états perdants")
    print(lp)
235
    """
237 jouer
    """
239 playAGame(5)
```

Exercices récursivité

Pelletier Sylvain PSI, LMSC

Exercice 1 Une autre méthode pour calculer les coefficients binomiaux

Écrire un algorithme qui permet de calculer récursivement la valeur de $\binom{n}{p}$ en utilisant :

$$\forall n \in \mathbb{N}^*, \forall p \in [\![1,n]\!], \qquad \binom{n}{p} = \frac{n}{p}\binom{n-1}{p-1}.$$

Déterminer le nombre d'appels en fonction de n et p.

Correction : Il ne faut pas oublier d'ajouter des conditions d'arrêt :

```
def binomRec(n,p) :
  if n<0 or p<0 or p>n :
    return 0
  elif n==0 or n==1 or p==0 or p==n:
    return 1
  else :
    return n/p*binomRec(n-1, p-1)
```

Normalement seul le cas particulier $p = 0$ est à considérer. Il n'y a que p appel à la fonction.
Par contre, on n'a pas un entier comme résultat.

Exercice 2 Générer les permutations

On considère $n \in \mathbb{N}^*$, on souhaite générer les $n!$ permutations de $[\![0,n-1]\!]$.
On note $\sigma(n)$ l'ensemble des permutation de $[\![0,n-1]\!]$ et on remarque que :

$$\sigma(n) = \bigcup_{k=0}^{n-1} S_k \qquad \text{union disjointe}$$

où S_k est l'ensemble des permutation s de $[\![0,n-1]\!]$, vérifiant $s(k) = n-1$.

De manière évidente, S_k est en bijection avec $\sigma(n-1)$, puisque pour construire une permutation de S_k, il faut placer les $n-1$ éléments de $[\![0,n-2]\!]$ dans les $n-1$ places différentes de k.

Autrement dit, chaque permutation s de $\sigma(n-1)$ donne n permutations de $\sigma(n)$: la première avec $n-1$ en première place, la deuxième avec $n-1$ en deuxième place, etc.

Écrire un algorithme récursif permettant de générer toutes les permutations en exploitant ce principe.

R On a retrouvé la formule : $n! = n \times (n-1)!$.

```
from pprint import pprint

def permutationRec(n):
    """
    entrée: n = int
    sortie: L = list = liste des n! permutations de
    [|0,n-1|]
    """
    if n == 1 :
        return [[0]]

    L = []
    for i in range(n-1, -1, -1) :
        for permut in permutationRec(n-1) :
            # on insère n-1 en position i
```

```
            permut.insert(i, n-1)
            L.append(permut)
    return L

n = 3
L = permutationRec(n)
print("les ",len(L)," permutations pour n=",n)
pprint(L)
```

Exercice 3 Soit n un entier naturel non nul, une **combinaison** de l'entier n est une liste $(n_1,\dots,n_p)$ d'entiers supérieurs ou égaux à 1 dont la somme fait n, *i.e.* telle que $n = n_1 + n_2 + \cdots + n_p$.

Une combinaison étant une liste, l'ordre y a de l'importance. Ainsi, la combinaison $(2,1)$, *i.e.* la décomposition $3 = 2+1$ diffère de la combinaison $(1,2)$, *i.e.* $3 = 1+2$.

Le but de cet exercice est de générer toutes les combinaisons d'un entier n.

Une combinaison sera représentée sous forme de liste et donc l'ensemble des combinaisons sous forme de liste de listes.

Pour $n = 1$, la seule combinaison est (1), l'ensemble des combinaisons est donc $\{(1)\}$, ce qui s'écrit en Python `[[1]]`.

Pour $n = 2$, les combinaisons sont (2) et $(1,1)$, ce qui s'écrit : $\{(2),(1,1)\}$, ou en Python : `[[2], [1,1]]`.

On remarque que l'on peut générer toutes les combinaisons de n de la manière suivante :

- Si $n = 1$ la seule combinaison est `[[1]]`
- Si $n > 1$, l'une des combinaisons est simplement n, et les autres sont obtenus à partir d'un entier $i \in [\![1, n-1]\!]$: on génère toutes les combinaisons de i et on ajoute $n-i$ à la fin pour obtenir une combinaison de n.

On obtient ainsi toutes les combinaisons de n à l'aide d'un algorithme récursif.

Programmer cet algorithme.

Pour vérifier :

- Les combinaisons de 3 sont : `[[3], [2,1], [1,2], [1,1,1]]`.
- Les combinaisons de 4 sont : `[[4], [3,1], [2,2] [1,3], [2,1,1], [1,2,1], [1,1,2], [1,1,1,1] ]`.
- Les combinaisons de 5 sont :

```
[[5], [4, 1], [3, 2], [3, 1, 1], [2, 3], [2, 2, 1], [2, 1, 2], [2, 1, 1, 1],
[1, 4], [1, 3, 1], [1, 2, 2], [1, 2, 1, 1], [1, 1, 3], [1, 1, 2, 1],
[1, 1, 1, 2], [1, 1, 1, 1, 1]]
```

Correction :

```
# -*- coding: utf-8 -*-
"""
génération des composition d'un entier n:
une composition est une liste n1, ... np telle que:
n1+n2+n3+ ... + np = n

Le principe est le suivant:
les combinaisons de n sont obtenues à partir des combinaisons de n-k
pour k dans  [|1, n-1|] en ajoutant k à la fin
"""


# Modules et fonctions importés
################################
from pprint import pprint # pour pouvoir bien afficher des listes.

# Fonctions
############
def compositionRec(n):
    """
    entrée: n entier = l'entier que l'on cherche à décomposer
    sortie: listeComposition = liste de listes d'entiers = la liste des décomposition
    renvoie la liste des composition de n
    """
```

```
    if n == 1 :
        return [ [1] ]
    listeComposition = [[n]]
    for i in range(1,n) :
        for compo in compositionRec(i):
            listeComposition.append(compo + [n-i])
    return listeComposition


n=5
print("la liste des composition pour n=",n)
pprint( compositionRec(n) )
```

Exercice 4 **Tours de Hanoi**

Les tours de Hanoï (originellement, la tour d'Hanoïa) sont un jeu de réflexion imaginé par le mathématicien français Édouard Lucas, et consistant à déplacer des disques de diamètres différents d'une tour de *départ* à une tour *d'arrivée* en passant par une tour *intermédiaire*, et ceci en un minimum de coups, tout en respectant les règles suivantes :

- on ne peut déplacer plus d'un disque à la fois ;
- on ne peut placer un disque que sur un autre disque plus grand que lui ou sur un emplacement vide.

Dans la position de départ, tous les disque sont regroupés sur la tour de départ dans l'ordre croissant de taille.

Le disque 1 au sommet, au dessus du disque 2, etc. jusqu'au disque n.

```
etat: [[1, 2, 3, 4], [], []]
   *|*         |           |
  **|**        |           |
 ***|***       |           |
****|****      |           |
```

Il faut obtenir la même configuration sur la tour d'arrivée.

```
etat: [[], [], [1, 2, 3, 4]]
    |           |         *|*
    |           |        **|**
    |           |       ***|***
    |           |      ****|****
```

Le but de cet exercice est d'écrire une fonction qui prend en entrée un entier n (représentant le nombre de disques) et qui calcule la liste des déplacements à effectuer sous la forme d'une liste de couple (i, j) de deux entiers distincts de 1 à 3. Cela signifie que l'on déplace le disque situé au sommet de la tour i à la tour j

La résolution de ce problème par récursivité est très simple :

- si il n'y a qu'un disque dans la position de départ, il faut le déplacer de la tour 1 à la tour 3.
- sinon, il faut :
  - déplacer les $n-1$ disques de la tour 1 à la tour 2 en se servant de la tour 3 comme intermédiaire ;
  - puis déplacer le dernier disque de la tour 1 à la tour 3
  - puis déplacer les les $n-1$ disques de la tour 2 à la tour 3 en se servant de la tour 1 comme intermédiaire.

La programmation récursive se fait donc via une fonction qui prend en entrée quatre arguments :

- Le nombre de disques à déplacer,
- la tour de départ, la tour d'arrivée et la tour intermédiaire.

Le prototype de cette fonction est :

```
def hanoiRec(n, D, A, I):
    """
    entrée: n =  int
              =  nbr de disques à déplacer
            D = int
              = indice de la tour de départ
            A = int = arrivée
            I = int = intermédiaire
    sortie : liste de couple de int de la forme (i,j)
            avec i!=j et (i,j) dans [|1,3|]
            liste des déplacements à effectuer
```

```
    """
```

La résolution du problème se fait via un simple appel à cette fonction récursive :

```
hanoiRec(n, 1, 3, 2)
```

Indication : vous disposez d'un script pour tester la résolution de votre exercice.

Correction :

```
def hanoiRec(n, D, A, I):
    """
    entrée: n =  int
              =  nbr de disques à déplacer
            D = int
              = indice de la tour de départ
            A = int = arrivée
            I = int = intermédiaire
    sortiie : liste de couple de int de la forme (i,j)
            avec i!=j et (i,j) dans [|1,3|]
            liste des déplacements à effectuer
    """
    if n == 1 :
        return [ [D,A] ]
    return hanoiRec(n-1, D, I, A) + [ [D,A]] + hanoiRec(n-1, I, A, D)


def resoud(n):
    """
    entrée: n = int
              = nbr de disques à déplacer
    sortie: liste de couple de int de la forme (i,j)
            avec i!=j et (i,j) dans [|1,3|]
            liste des déplacements à effectuer
    """
    return hanoiRec(n, 1, 3, 2)


def test(n, listeD):
    """
    entrée: n = int
              = nbr de disques à déplacer
            listeD liste de couple de int de la forme (i,j)
            avec i!=j et (i,j) dans [|1,3|]
            liste des déplacements à effectuer
    sortie: rien
    """
    etat = [ [i+1 for i in range(n)], [], []]
    represente(etat, n)
    for i,j in listeD :
        print("déplacement d'un disque de la tour", i," vers la tour ", j)
        disque = etat[i-1].pop(0)
        etat[j-1].insert(0, disque)
        represente(etat, n)
        assert verifie(etat)
    print("tout est ok")

def verifie(etat):
    """
    entrée: etat = liste de 3 liste de int
                 = etat du jeu
              = nbr de disques à déplacer
    sortie True / False
    """
    for tour in etat :
```

```
        n = len(tour)
        for i in range(n-1):
            if tour[i] >= tour[i+1]:
                return False
    return True

def represente(etat,n ):
    """
    entrée: etat = liste de 3 liste de int
                 = etat du jeu
            n = int
    sortie: rien affichage
    """
    print("etat:", etat)
    tour = [ [" "*n +"|" + " "*n   for i in range(n)] for j in range(3)]
    for j in range(3):
        for l in range(len(etat[j])) :
            l2 = l + n- len(etat[j])
            r = etat[j][l]
            tour[j  ][l2] = " "*(n - r)+"*"*r + "|" + "*"*r +" "*(n - r)
    tourTotale = [ tour[0][i] + " " + tour[1][i] + " " + tour[2][i] for i in range(n)]
    for l in range(n):
        print(tourTotale[l])


print("---- test de l'algo récurisif des tours de Hanoi ----")
n =  4
listeD = resoud(n)
print("liste des déplacements:", listeD)
test(n, listeD)
```

2 — Bases de données

Conformément au programme, on se limite à une description applicative des bases de données en langage SQL.

Il s'agit simplement de donner des rudiments du langage des requêtes pour interroger une vase présentant des données à travers plusieurs relations.

On ne présente pas l'algèbre relationnelle, ni le calcul relationel.

I Le problème de la gestion de l'information

I.1 Introduction

L'informatique est la science de l'information et de son traitement. En conséquence, en informatique on traite un grand nombre de données.

On peut citer quelques cas extrêmes :

- Les 3 milliards de compte facebook (actif),
- les 20 millions de parties de go nécessaire à AlphaGo.
- Les milliards de page web indexés par Google.

Dans un registre différent, on peut citer le système de la SNCF pour la réservation des trains ou l'ensemble des ordres bancaires dûs aux achats par carte bleue.

Il faut bien comprendre les problèmes posés par la gestion de l'information :

- On a beaucoup de données, ces données sont hétérogènes,
- on veut y accéder rapidement, (il faut gagner des microsecondes), de plus de différentes manières,
- avec un système mutli-utilisateur sécurisé (tous les utilisateurs n'ont pas les mêmes droits),
- et un risque d'écriture simultanées, une gestion des pannes, etc.

Tout ceci est un métier à part entière. On va juste dans ce chapitre exposer les concepts et le vocabulaire de base, puis des rudiments de langage de requête.

Au concours, il y a souvent des petites questions de MySql qui concernent

souvent une agrégation ou des jointures.

■ **Exemple I.1** On peut considérer un système informatique pour collecter les notes de colles.

On veut pourvoir accéder aux notes d'un élève, d'un colleur, d'une matière, d'un élève dans une matière, etc. Il faut donc un système de recherche multi-critère.

On a un aspect multi-utilisateur : un élève ne doit pas pouvoir modifier une note. ■

■ **Exemple I.2** On prendra aussi l'exemple d'une bibliothèque de prêt.

Les objets sont les livres et les abonnés, ainsi que les prêts des livres. ■

■ **Exemple I.3** Le dernier exemple que l'on suivra sera un site de vente, avec une table pour les clients, une table pour les objets et une table pour les achats ■

L'ensemble de cette problématique de gestion de l'information a été étudié dans les années 60 et a donné naissance aux bases de données relationnelles.

I.2 Vocabulaire des bases de données

Pour traiter cette information, on voit bien que l'on ne peut pas utiliser un tableau à deux dimensions (tableur), puisqu'il faut traiter de l'information hétérogène.

Il faut donc organiser différemment l'information. En fait, on va utiliser plusieurs tableaux et les relier entre eux.

Le modèle que l'on va étudier pour représenter l'information est d'utiliser plusieurs tableaux à deux dimensions (une **table de données**)

Une table de données est un tableau stockant de l'information sur un type d'objets. Elle contient des enregistrements ou des lignes : les caractéristiques d'un objet.

Une base de données est donc un ensemble de table de données.

Les attributs d'un objet sont les caractéristiques d'un objet, c'est donc les colonnes de la table. Pour un objet donné, les attributs sont déterminés de manière unique.

À un attribut est associé un domaine c'est l'ensemble des valeurs que peut prendre cet attribut (c'est le type de données).

Le schéma de la table est le produit cartésien des domaines des attributs.

Pour le type de domaine, on peut avoir : des entiers, des flottants, des chaînes de caractères, des dates, etc.

Aucune notion sur les dates n'est au programme (pourtant elles sont très utilisées). Il y a plein de méthodes pour comparer les chaînes de caractères, mais là aussi c'est hors programme.

Un attribut peut être `NULL` (hors-programme).

Ce découpage de l'information permet de la manipuler plus rapidement : en effet, si on veut ajouter un sujet ou modifier les coordonnées d'un autre, il n'y a qu'une table à modifier.

■ **Exemple I.4** Pour les notes de colles, on a va avoir plusieurs tables :

- La table élèves, dont les attributs sont : nom (chaîne), prénom (chaîne), email (chaîne), date de naissance (date). On ajoute généralement un identifiant unique (entier).

- La table des professeurs : nom (chaîne), prénom (chaîne), matière (chaîne), email (chaîne) et encore un identifiant unique.
- La table des groupes : identifiant (int), identifiant élève1 (int), identifiant élève2 (int), identifiant élève3 (int),
- La table des notes : identifiant de l'élève, identifiant du professeur, date-heure (date).
- La table des colles : identifiant du professeur (int), identifiant du groupe (int), numéro de semaine (int), salle (chaîne), date-heure (date).

■

Les relations entre les objets se font à travers l'utilisation d'identifiant unique : les clés primaires. C'est un sous-ensemble d'attributs qui identifient de manière unique un objet.

Souvent, on ajoute un identifiant unique, qui sert de clé primaire.

■ **Exemple I.5** Dans les tables élèves et professeurs, c'est cet identifiant que l'on utilise.

Dans la table colle, c'est le triplet (id groupe, id professeur, numéro de semaine) que l'on peut choisir.

■

Les clés primaires permettent de faire très rapidement des recherches.

Ces objets interagissent entre eux. La grande idée des bases de données est de stocker les relations entre objets comme des entités, puis de faire des relations entre objets et entités.

On parle du modèle **entité-relation** pour expliquer que l'on stocke dans les tables soit des objets (entités), soit de relations entre objets.

On parle de clé étrangère (dans la table 2) lorsqu'une clé primaire d'une table 1 est utilisée comme attribut dans la table 2.

■ **Exemple I.6** Par exemple, pour une bibliothèque, la table prêt contient les relations entre les objets utilisateurs et les objets livres. Pour la table prêt, les identifiants du livre et de l'utilisateur sont des clés étrangères.

Livres
id (int)
titre (chaîne)
idauteur (int)
nbrPages (int)

Auteurs
id (int)
nom

Utilisateurs
id (int)
nom (chaîne)
prénom (chaîne)
email (chaîne)

Emprunts
idUtilisateur (int)
idLivre (int)

■

Généralement, on représente l'organisation de l'information en table sous la forme de diagramme UML.

■ **Exemple I.7** Pour les notes de colles :

élèves
id (int)
nom (chaîne)
prénom (chaîne)
email (chaîne)
date de naissance (date)

professeur
id (int)
nom (chaîne)
prénom (chaîne)
matière (chaîne)
email (chaîne)

groupes
id groupe (int)
id élève 1 (int)
id élève 2 (int)
id élève 3 (int)

notes
id élève (int)
id professeur (int)
date et heure (date)

colles
id professeur (int)
id groupe (int)
salle (chaîne)
numéro de semaine (int)
date et heure (date)

(faire des flèches entre les clé étrangères). ■

■ **Exemple I.8** Pour le site de vente :

Clients
id (int)
nom (chaîne)
prénom (chaîne)
email (chaîne)
parrain (id)

Objets
id (int)
nom (chaîne)
prix (float)

Achats
idClient
idObjet
date (date)
quantite (int)

■

Lorsqu'on manipule une base de données, il faut toujours avoir sous les yeux le schéma de la base (les différentes tables et leur lien). Il faut aussi penser à noter le type de données (le domaine des attributs).

✯ La solution matérielle : le modèle client-serveur

Puisqu'on veut partager de l'information entre plusieurs utilisateurs, il faut donc deux machines :

- une machine qui stocke les données : **le serveur**. Sur ce serveur est installé un **système de gestion de base de données**. Il n'y a qu'un serveur.
- Une machine **client** pour envoyer des ordre de recherche ou de modification d'une base de données qui sont transmises au serveur.
  !il peut y avoir plusieurs clients.

C'est l'architecture client-serveur.

> On parle de requêtes pour désigner ces ordres de demande d'information envoyés au serveur de données.

Pour créer les système de gestion de base de données, il y a eu tout d'abords une étude théorique (l'algèbre relationnelle faite par les mathématiciens), puis les industriels se sont décidés pour un langage standard (SQL « standard query langage ») de requêtes. Ce langage est donc un standard qui n'appartient à aucune entrprise.
C'est ce langage de requête que l'on doit étudier dans ce cours.
Ensuite, des logiciels de gestion de base de données ont été écrits capable de

répondre à des requêtes (écrites dans ce langage standard). Ces logiciels peuvent appartenir à des entreprises.
Parmi ces logiciels de gestion, on peut citer :
- Oracle (payant très cher),
- MySql (libre et gratuit très puissant),
- SqLite (libre et gratuit, un peu plus limité)

En pratique, on utilise le plus souvent une architecture trois-tiers :
- L'utilisateur final interagit avec un logiciel client, cela peut être un logiciel (écrit par exemple en Python) ou une page web dans le navigateur (écrite en Php),
- Cette application crée une requête (lecture ou écriture sur une base de données) qui est envoyé au serveur.
- Le serveur répond et renvoie l'information à l'application, qui l'affiche.

■ **Exemple I.9** Lorsque l'on consulte un site de vente en ligne, on a trois niveaux :
- le navigateur internet installé sur l'ordinateur personnel de la personne qui consulte le site de chez elle, envoie une demande pour consulter des données, par exemple la liste des articles à moins de 30 euros. Cette demande est faite par le remplissage d'un formulaire.
- Cette demande est transmise par le formulaire HTML au serveur web, qui la transforme en requête pour le serveur de données,
- le serveur de données envoie les informations au serveur web, qui construit la page web correspondante,
- enfin, cette page s'affiche sur le navigateur internet.

■

II Requêtes de lecture dans une table

II.1 Projection

Pour lire de l'information dans une table, on utilise le mot clé `SELECT`. La syntaxe est :

```
SELECT att1, att2 FROM table;
```

où `att1` et `att2` sont les les caractéristiques des objets de la table les attributs qui nous intéressent. On dit que l'on projette sur ces attributs.

Si tous les attributs nous intéressent, on utilise :

```
SELECT * FROM table;
```

Si l'on souhaite trier, on utilise :

```
--tri croissant par att3
SELECT att1, att2 FROM table ORDER BY att3;
-- tri décroissant par att3
SELECT att1, att2 FROM table ORDER BY att3 DESC;
```

Si l'on veut effectuer une troncature (ne garder que quelques éléments) :

```
-- ne garder que les 5 premiers éléments
SELECT att1, att2 FROM table
            ORDER BY att3 LIMIT 5;
-- ne garder que les  éléments de 10 à 30
SELECT att1, att2 FROM table
            ORDER BY att3 LIMIT 20 OFFSET 10;
```

Si l'on veut ne garder que les valeurs distinctes d'un attribut, on utilise `DISTINCT` :

```
SELECT DISTINCT att1 FROM table;
```

Voici quelques exemples :

```
SELECT titre, auteur FROM livres;
SELECT * FROM livres; # sélection de tous les attributs
-- pour le tri
SELECT titre, auteur FROM articles ORDER BY annee;
SELECT titre, auteur FROM articles  ORDER BY titre DESC;
SELECT titre, auteur
FROM articles  ORDER BY titre DESC, auteur;
-- pour la troncature
-- 10 premiers résultat
SELECT titre, auteur
FROM articles ORDER BY annee LIMIT 10;
-- résultat 20 à 30
SELECT titre, auteur
FROM articles ORDER BY annee LIMIT 10 OFFSET 20;
-- deuxième résultat
SELECT titre, auteur
FROM articles ORDER BY annee LIMIT 1 OFFSET 1;
```

II.2 Sélection

La sélection consiste à extraire de la table les éléments qui vérifient une condition

Une sélection consiste à ne garder que les éléments qui vérifient une condition (les autres sont effacés). La sélection se fait avec le mot clé `WHERE`.

La syntaxe est :

```
SELECT * FROM Table WHERE condition;
```

on peut bien sûr remplacer `*` par la liste des attributs qui nous intéresse (sélection suivi d'une projection).

Les conditions possibles sont constuites avec `+`, `-`, `*`, `/` (on passe outre les subtilités liées à la division entière ou flottante), `=`, `<>`, `<`, `<=`, `>`, `>=`, `AND`, `OR`, `NOT`.

Étant donné leur utilisation massive, on utilise un format spécial pour les dates et les heures qui permet d'effectuer rapidement des comparaisons. De même, on peut très rapidement comparer des chaînes de caractères.
Tout ceci est hors-programme.
Extrait du programme : *Aucune notion relative à la représentation des dates n'est*

au programme ; en tant que de besoin on s'appuie sur des types numériques ou chaîne pour lesquels la relation d'ordre coïncide avec l'écoulement du temps. Toute notion relative aux collations est hors programme ; on se place dans l'hypothèse que la relation d'ordre correspond à l'ordre lexicographique usuel.

```
SELECT * FROM livres WHERE annee < 1850;
```

II.3 Renommage

le **renommage** consiste à changer le nom d'un attribut.

Le mot clé pour le renommage s'appelle `AS` suivi par le nouveau nom de l'attribut.

La syntaxe est :

```
SELECT oldName1 AS newName1, oldName2 AS newName2
FROM nomTable WHERE condition ;
```

R Le renommage a un intérêt pratique par exemple lorsqu'on mélange les tables.

III Requêtes de lecture utilisant deux tables

III.1 Opérateur ensembliste

Soit `table1` et `table2` deux tables, basées sur le même schéma.

On peut alors utiliser :

```
SELECT * FROM table1 UNION table2;
SELECT * FROM table1 INTERSECT table2;
SELECT * FROM table1 MINUS table2;
```

■ **Exemple III.1** Cela revient à chercher dans deux bibliothèques. ■

R Dans le langage MySQL, `UNION` existe, mais ni `INTERSECT`, ni `MINUS`.

Si un élément est présent dans les table 1 et 2, on peut distinguer `UNION ALL` qui garde les deux éléments, et `UNION DISTINCT` (ou simplement `UNION`) qui n'en garde qu'un.

III.2 Jointure

Les jointures permettent de mélanger de relations (des tables) et donc de faire des requêtes croisées. La jointure la plus simple (et en pratique très peu utilisée) est le **produit cartésien** : à partir d'une table `tab1` de n objets notés (x_i) d'une autre table `tab2` de m objets notés (y_j), on construit une liste de $n \times m$ objets de la forme $\left(x_i, y_j\right)$ pour $i \in [\![1,n]\!]$ et $j \in [\![1,m]\!]$.

La syntaxe consiste simplement à écrire les deux tables :

```
SELECT * FROM table1, table2 ;
```

■ **Exemple III.2** On peut considérer le produit cartésien des clients et des achats :

```
SELECT clients.id, clients.nom, commandes.id_client,
commandes.id_objet, commandes.quantite
FROM clients, commandes ;
```

Cela crée une liste des nom des clients et des commandes (qui peuvent être faits par d'autres clients). Cela n'a bien sûr aucun intérêt en pratique.

Par contre, si on filtre en ne gardant que les lignes où l'identifiant client (dans la table clients) est celle de l'identifiant client dans la table commandes :

```
SELECT clients.id, clients.nom, commandes.id_client,
commandes.id_objet, commandes.quantite
FROM clients, commandes
WHERE commandes.id_client = clients.id ;
```

Alors, là on obtient ce qui est attendu : chaque ligne corresponds à une commande d'un client. ■

✯ La jointure symétrique

On voit sur l'exemple que ce qui est intéressant en pratique c'est de faire un produit cartésien suivi d'une sélection en ne gardant que les lignes où un attribut est égal dans deux tables (très souvent une clé étrangère).

C'est le principe d'une jointure symétrique que l'on va utiliser.

La jointure symétrique consiste à combiner deux tables en ne gardant que les éléments qui ont une caractéristique en commun.

Très souvent c'est une clé primaire qui est identique.

La syntaxe est :

```
SELECT attributs FROM table1
JOIN table2 ON table1.att1 = table2.att2
```

La liste obtenue est alors le croisement entre la table 1 et la table 2.

■ **Exemple III.3** Pour retrouver les commandes par clients, on peut faire :

```
SELECT clients.nom, commandes.id_objet, commandes.quantite
FROM clients
JOIN commandes ON commandes.id_client = clients.id
```

Si on veut retrouver les commandes faits par le client 3141, il suffit de faire :

```
SELECT clients.nom, commandes.id_objet, commandes.quantite
FROM clients
JOIN commandes ON commandes.id_client = clients.id
WHERE clients.id = 3141
```

Si on veut les noms des objets et non plus leur identifiant, il faut faire deux jointures symétriques :

```
SELECT clients.nom, objets.nom, commandes.quantite
FROM clients
JOIN commandes ON commandes.id_client = clients.id
JOIN objets ON commandes.id_objets = objets.id
```

■

⋆ Autojointure

Il peut arriver que l'on jointe une table avec elle-même. C'est le cas lorsque la table contient un attribut qui est en fait sa propre clé primaire.

La syntaxe consiste à renommer la table pour faire la jointure avec elle-même :

```
SELECT *
FROM table
JOIN table AS ext ON table.att = ext.id
```

R On voit ici l'intérêt du renommage.

■ **Exemple III.4** Un exemple parlant est une table contenant une liste d'animaux d'élevage, où on garde la trace des parents de l'animal.

On peut alors faire :

```
SELECT Animal.nom AS nomAnimal,
       Pere.nom AS nomPere,
       Mere.nom AS nomMere
FROM Animal
JOIN Animal AS Pere ON Animal.pere_id = Pere.id
JOIN Animal AS Mere ON Animal.mere_id = Mere.id
```

■

⋆ Vocabulaire cardinalité d'une association

Comme on l'a vu précédemment, le principe abstrait des bases de données est de stocker les objets (**entités**) dans des tables séparées et de stocker les **relations** (ou **d'associations**) entre ces objets dans des tables aussi, comme des objets.

Par exemple : les objets « livres » et les objets « utilisateurs » sont en relation (puisque les utilisateurs empruntent des livres). Cette association est stockée dans la table emprunts.

Lorsque l'on a une telle relation, on regarde combien de fois les objet peut être présent dans chaque relation. On parle de la `cardinalité` d'une association.

Par exemple pour les livres, on peut considérer que la relation entre livres et auteur est de la forme :

```
Livre --- (1,1) --- est écrit par --- (0,n) --- auteurs
```

Les chiffres indiqués entre parenthèses indiquent combien de fois chaque objet est présent au minimum et au maximum dans la relation. Ainsi ici :

- Un livre a minimum un auteur et maximum un auteur (on fait le choix de négliger les ouvrages écrits par plusieurs personnes).
- Un auteur peut avoir écrit aucun livre ou une infinité (le n indique un nombre quelconque).

R Dans le programme, on ne regarde les cardinalités que pour le maximum. C'est à dire que l'on ne garde que les maximum de chaque côté. On écrit donc `1-*` ou

`1-n` pour indiquer qu'un livre a au plus un auteur tandis que les auteurs peuvent écrire un nombre quelconque de livres.

En MySql, on a donc souvent le cas où il y a trois tables, `table1` et `table2` contiennent les caractéristiques des objets (entités), et `table3` contient les relations entre ces objets (association).

Dans `table1` et `table2` on a des clés primaires (appelées `cle1` et `cle2`). Ces clés primaires sont présentes comme attributs dans la table `table3`, on parle donc de clés étrangères.

On regarde alors combien de fois (au moins et au plus) peut être présente une clé `cle1` dans la table `table3`. Par exemple :

- $(0,1)$ 0 fois au minimum, 1 fois au maximum,
- $(0,n)$ 0 fois au minimum, autant de fois au maximum,
- $(1,n)$: 1 fois au minimum, autant de fois au maximum,
- $(1,1)$: 1 et une seule fois.

C'est la cardinalité de la relation entre les objets de la `table1` et ceux de la `table2`.

IV Agrégations et fonctions d'agrégation

L'agrégation consiste à regrouper des données ayant une caractéristique commune dans le but d'y appliquer des fonctions statistiques. C'est-à-dire de remplacer un ensemble d'objets enregistrés que l'on considère comme équivalent (ils forment un lot, un tas, un agrégat) et d'extraire de ce regroupement des caractéristiques qui dépendent de cet ensemble d'objets.

⋆ Fonctions d'agrégation

Les fonctions d'agrégation s'appliquent à une colonne complète, c'est à dire à un attribut d'une table, elle permettent de créer une nouvelle valeur.

La syntaxe est

```
SELECT FCT(attribut) FROM table
```

On peut utiliser : `FCT(attribut)` ou `FCT(attribut1, attribut2)` ou encore `FCT(*)`.

Dans le programme, il y a quatre fonctions d'agrégations :

Minimum et maximum avec `MAX` et `MIN` respectivement,

Le comptage avec `COUNT`

La somme et la moyenne avec `SUM` et `AVG`

Du point de vue mathématique, une fonction d'agrégation s'applique à un nombre quelconque d'argument (ie à une colonne de taille quelconque) et ne dépende pas de l'ordre des éléments.

Ces fonctions servent à faire des statistiques sur les données.

■ **Exemple IV.1** Pour avoir le prix le plus cher et le prix moyen :

```
SELECT max(prix), AVG(prix) FROM objets
```

```
SELECT countmax(prix), AVG(prix) FROM objets
```

■

✯ Agrégation

Avant d'appliquer une fonction d'agrégation, on peut appliquer une agrégation : on regroupe ensemble les objets ayant un même attribut (une caractéristique commune). On obtient ainsi un agrégat, c'est-à-dire un ensemble d'objet qui ont un attribut en commun. L'intérêt est d'appliquer une fonction d'agrégation sur les données regroupées.

Cela est fait avec le mot clé `GROUP BY` suivi de l'attribut qui sert d'agrégat. La syntaxe la plus simple est :

```
SELECT att1, F(att2)
FROM table
GROUP BY att1
```

On regroupe alors les objets ayant la même valeur de `att1` puis on applique sur ce groupe la fonction d'agrégation `F`.

En plus des fonctions d'agrégation, on peut projeter sur l'attribut sur lequel on fait l'agrégation, ou sur un attribut défini de manière unique en fonction de cet attribut. Par exemple dans la syntaxe :

```
SELECT att1, att2,  F(att3)
FROM table
GROUP BY att1
```

il faut que si deux objets ont la même valeur pour `att1`, alors ils ont la même valeur pour `att2`.

On peut aussi regrouper selon deux attributs :

```
SELECT att1, att2,  F(att3)
FROM table
GROUP BY att1, att2
```

Les groupes sont alors faits sur les objets qui ont la même valeur pour les deux attributs `att1` et `att2`.

■ **Exemple IV.2** Nombre de commandes par jour :

```
SELECT date, count(*) FROM commandes GROUP BY date
```

■

■ **Exemple IV.3** Si on veut calculer le total des achats de chaque client :

```
SELECT clients.nom as nomClient,
       clients.email as emailClient,
       SUM(objets.prix * achats.quantite) as totalAchat
FROM clients
JOIN achats ON achats.idClient=clients.idClient
JOIN objets ON achats.idObjet=objets.idObjet
GROUP BY clients.id
```

■

✯ Sélection après agrégation

Lorsque l'on a regroupé les objets et appliquer une fonction d'agrégation, on peut éventuellement choisir de sélectionner que les données vérifiant une condition. Cette sélection est faite après agrégation et après application des fonctions d'agrégation. En particulier, on pourra faire dépendre cette condition du résultat des fonctions d'agrégation. Le mot clé est `HAVING` suivi de la condition.

La syntaxe la plus courante est donc :

```
SELECT att1, F(att2) AS valeur
FROM table
GROUP BY att1
HAVING valeur >= x
```

Le traitement se fait alors ainsi :

- on regroupe alors les objets selon la valeur de `att1`,
- puis on applique la fonction d'agrégation `F`, ce qui permet d'attribuer une valeur à chacun des groupes.
- Enfin, on ne garde que les groupes où cette valeur est supérieure à `x`.

Il faut comprendre que l'on fait alors une selection sur les groupes et non sur les objets. Les calculs sont faits quand même sur toutes les données, simplement, les groupes qui ne satisfont pas la condition ne sont pas affichés. En particulier, l'agrégation est faite sur toutes les données. Au contraire, la sélection faite avant l'agrégation (avec `WHERE`) consiste à « effacer » les données qui ne vérifient pas la condition. Aucun calcul n'est alors fait sur ces données.

■ **Exemple IV.4** Dans le but d'envoyer un email aux clients ayant fait plus de 200 euros d'achat depuis le premier septembre :

```
SELECT clients.nom as nomClient,
       clients.email as emailClient,
       SUM(objets.prix * achats.quantite) as totalAchat
FROM clients
JOIN achats ON achats.idClient=clients.idClient
JOIN objets ON achats.idObjet=objets.idObjet
WHERE achats.date > "2022-09-01"
GROUP BY clients.id
HAVING totalAchat >= 200
```

■

V Séquence complète en SQL

En conclusion, la séquence complète SQL est :

```
SELECT att, f(atts)
FROM table1
JOIN table2 ON table1.idx = table2.idx
WHERE condition
GROUP BY att
```

```
HAVING condition2
```

On doit donc décider dans l'ordre :

- Sur quelle donnée on veut travailler : quelle table veut-on utiliser (`FROM`) ? Est-ce que l'on veut croiser cette table avec une autre table (`JOIN ... ON ...`) ? Est-ce que l'on veut utiliser la table complète ou uniquement les données vérifiant une condition (`WHERE`) ?
- Est-ce que les données doivent être regroupées selon un ou plusieurs attributs dans le but d'en extraire de l'information statistique ?
- Quels sont les caractéristiques (attributs) que l'on veut récupérer (`SELECT`) ? quelles sont les statistiques sur les données regroupées (fonction d'agrégations) que l'on veut ?
- Éventuellement, on peut choisir de ne pas afficher tous les résultats mais uniquement ceux vérifiant une condition (`HAVING`)

Introduction aux bases de données

Pelletier Sylvain PSI, LMSC

Pour pouvoir tester les requêtes, nous allons utiliser une base données qui a été déposée en ligne sur le serveur (gratuit) du site `alwaysdata`. Nous allons utiliser l'interface en ligne `phpmyadmin` pour interroger le serveur de base de données.

- Depuis internet, se rendre sur le site : `https://phpmyadmin.alwaysdata.com`
- Rentrer l'identifiant :
- Rentrer le mot de passe :
- Sélectionner la base de données : `pscilmsc_transport` (onglet de gauche).
- Sur la partie centrale onglet `structure`, se familiariser avec l'interface. en particulier `structure` et `parcourir`.
- Pour écrire une requête cliquer sur l'onglet `SQL`. Cocher conserver la boite de requête puis écrire la requête.
- Il est très pratique d'utiliser un bloc note et de copier / coller du bloc note vers `phpmyadmin`.

V.1 Description de la base de données

La base contient les tables suivantes (la clé primaire est soulignée) :

☞ Table **régions** :
- `r_id` : tinyint,
- `nom` : varchar(50),
- `préfecture` : varchar(5),
- `population` : int,
- `superficie` : int,
- `orientation` : varchar(10),
- `littoral` : tinyint,
- `frontières` : tinyint

☞ Table **villes** :
- `code_postal` : varchar(5),
- `nom` : varchar(50),
- `région` : tinyint,
- `population` : int,
- `Latitude` : decimal
- `Longitude` : decimal

☞ Table **autoroute** :
- `numérotation` : smallint,
- `concessionnaire` : varchar(5),

☞ Table **réseau** (= autoroutes) :
- `ville` : varchar(5),
- `autoroute` : smallint,
- `position` : tinyint,
- `distance` : smallint

☞ Table **trajets** (= train) :
- `Depart` : varchar(5)
- `Arrivee` : varchar(5),
- `Durée` : time,
- `Prix` : tinyint,
- `TGV` : tinyint

Clés étrangères :
- `préfecture` est une clé étrangère **régions** référençant `code_postal` de **villes**.
- `région` est une clé étrangère **villes** référençant `r_id` de **régions**.
- `ville` est une clé étrangère **réseau** référençant `code_postal` de villes.
- `Départ` et `Arrivées` ont des clés étrangères **trajets** référençant `code_postal` de **villes**.
- `autoroute` est une clé étrangère **réseau** référençant `numérotation` de **autoroutes**.

Q1 Tracer le schéma relationnel des tables, des clés primaires et des clés étrangères qui les relient.

Les questions suivantes consistent à écrire et lancer les requêtes correspondant à la question posée. Ecrire la requête sous chaque question.

VI Requête simple sur une table

Q2 Afficher les 5 premières lignes de la table villes, puis les lignes 5 à 10. Quel est l'ordre choisi par mysql ? L'ordre est celui du premier attribut par ordre croissant

```
SELECT * FROM villes LIMIT 5
```

```
SELECT * FROM villes LIMIT 5 OFFSET 4
```

Q3 Donner le nom des régions possédant un littoral ?

```
SELECT nom FROM régions WHERE Littoral = 1
```

Q4 Donner le nom des régions situées dans la moitié sud du pays ?
Indication : Orientation contient IDF, Nord-Est, Nord-Ouest, Sud-Est ou Sud-Ouest

```
SELECT nom FROM régions WHERE Orientation LIKE "Sud%"
SELECT nom FROM régions WHERE Orientation = "Sud-Ouest" OR Orientation = "Sud-Est"
SELECT nom FROM régions WHERE Orientation IN  ( "Sud-Ouest", "Sud-Est")
```

Q5 Donner le nom et la population des régions ayant au moins une frontière avec un autre pays et ranger ces régions par ordre décroissant de population.

```
SELECT nom,population,Frontières FROM régions
WHERE Frontières > 0
ORDER BY population DESC
```

Q6 Donner la population totale vivant dans une région ayant un littoral.

```
SELECT SUM(population) FROM régions WHERE Littoral=1
```

Q7 Donner la population moyenne des régions en France

```
SELECT AVG(population) FROM régions
```

Q8 Combien y a-t-il de villes situées à l'ouest de Paris ?
Indication on sait rechercher la longitude de la ville de Paris, et on va utiliser ce résultat (qui est un nombre réel) dans une condition, c'est une requête multiple.

```
SELECT count(*) FROM villes
WHERE Longitude<(SELECT Longitude from villes WHERE nom = "Paris")
```

Q9 Donner pour chaque région son nom, sa population, sa superficie et sa densité de population. On les ordonnera par ordre alphabétique.

```
SELECT nom, population, superficie,
        population/superficie AS densité_de_population FROM régions ORDER BY nom
```

VII Jointures simples

Q10 Donner les noms de chaque région et de leur préfecture ainsi que la population de leur préfecture

```
SELECT régions.nom,villes.nom,villes.population FROM régions
JOIN villes ON villes.code_postal = régions.préfecture
```

Q11 Donner les noms et populations des villes de Rhône Alpes

```
SELECT régions.nom,villes.nom,villes.population FROM régions
JOIN villes ON régions.r_id=villes.région
WHERE régions.nom="Rhône Alpes"
```

Q12 Donner le nom des villes et la population des villes traversées par l'autoroute A4.

```
SELECT villes.nom, villes.population FROM réseau
JOIN villes ON réseau.ville=villes.code_postal
WHERE réseau.autoroute=4
```

Q13 Donner tous les trajets en train impliquant la ville de Nantes.

```
SELECT trajets.* FROM `trajets`
JOIN villes AS vd ON vd.code_postal = trajets.depart
JOIN villes AS va ON va.code_postal = trajets.depart
WHERE vd.nom = 'Nantes' OR va.nom = 'Nantes';
```

VIII Jointures multiples

Q14 Donner les régions traversées par l'autoroute A4.

```
SELECT DISTINCT régions.nom FROM réseau
JOIN villes ON réseau.ville=villes.code_postal
JOIN régions ON villes.région=régions.r_id
WHERE réseau.autoroute=4
```

Q15 Donner le nom des régions traversées par des autoroutes de VINCI.

```
SELECT DISTINCT réseau.autoroute FROM réseau
JOIN autoroute ON autoroute.Numérotation=réseau.autoroute
JOIN villes ON réseau.ville=villes.code_postal
JOIN régions ON villes.région=régions.r_id
WHERE autoroute.Concessionnaire="VINCI"
```

Q16 Donner les trajets en train reliant deux villes de plus de 150000 habitants.

```
SELECT v1.nom,v2.nom,trajets.duree, trajets.prix,trajets.TGV FROM trajets
JOIN villes AS v1 ON trajets.depart=v1.code_postal
JOIN villes AS v2 ON trajets.arrivee=v2.code_postal
WHERE v1.population >= 150000 AND v2.population >= 150000
```

Q17 Donner les trajets en train reliant une ville de Bretagne à une ville de Rhône-Alpes.

```
SELECT t.*
FROM trajets AS t
JOIN villes AS v1 ON t.depart=v1.code_postal
JOIN villes AS v2 ON t.arrivee=v2.code_postal
JOIN régions AS r1 ON v1.région=r1.r_id
JOIN régions AS r2 ON v2.région=r2.r_id
WHERE (r1.nom='Bretagne' AND r2.nom='Rhône Alpes')
   OR (r1.nom='Rhône Alpes' AND r2.nom='Bretagne')
```

IX Agrégation

Q18 Pour chaque secteur du pays (orientation) donner la population totale de ce secteur.

```
SELECT Orientation, SUM(population) FROM régions
GROUP BY Orientation
```

Q19 Pour chaque autoroute, donner sa longueur.

```
SELECT autoroute,max(distance) FROM réseau
GROUP by autoroute
```

Q20 Donner pour chaque région, le nombre de villes présentes.

```
SELECT régions.nom,count(villes.nom) FROM régions
JOIN villes ON régions.r_id=villes.région
GROUP BY régions.nom
```

Q21 Pour chaque secteur (orientation), le nombre de villes de chaque secteur.

```
SELECT régions.orientation,count(villes.nom) FROM régions
JOIN villes ON régions.r_id=villes.région
GROUP BY régions.orientation
```

Q22 Donner pour chaque autoroute, le nombre de villes qu'elle relie.

```
SELECT autoroute, count(ville) FROM réseau GROUP BY autoroute
```

Exemples de requêtes MySQL

Pelletier Sylvain PSI, LMSC

Dans cette fiche on montre les exemples à connaître de requêtes sur une base de données. Pour illustrer ces requêtes, on considère une base de données de taille moyenne concernant un élevage commercial d'animaux.

Pour pouvoir tester les requêtes, nous allons utiliser une base données qui a été déposée en ligne sur le serveur (gratuit) du site `alwaysdata`. Nous allons utiliser l'interface en ligne `phpmyadmin` pour interroger le serveur de base de données.

- Depuis internet, se rendre sur le site : `https://phpmyadmin.alwaysdata.com`
- Rentrer l'identifiant :
- Rentrer le mot de passe :
- Sélectionner la base de données : `psilmsc_test` (onglet de gauche).
- Sur la partie centrale onglet `structure`, se familiariser avec l'interface. en particulier `structure` et `parcourir`.
- Pour écrire une requête cliquer sur l'onglet `SQL`. Cocher conserver la boite de requête puis écrire la requête.
- Il est très pratique d'utiliser un bloc note et de copier / coller du bloc note vers `phpmyadmin`.

Cette base de données fictive provient du cours : Administrez vos bases de données avec MySQL, disponible sur le site de site openclassrooms.

Cette base de données est constituée ainsi :

```
SHOW TABLES;
+------------------+
| Tables_in_testPC |
+------------------+
| Adoption         |
| Animal           |
| Client           |
| Espece           |
| Race             |
+------------------+
```

La table `Animal` contient la liste des animaux dans le magasin :

```
DESCRIBE Animal;
+----------------+----------------------+------+-----+---------+----------------+
| Field          | Type                 | Null | Key | Default | Extra          |
+----------------+----------------------+------+-----+---------+----------------+
| id             | smallint(6) unsigned | NO   | PRI | NULL    | auto_increment |
| sexe           | char(1)              | YES  |     | NULL    |                |
| date_naissance | datetime             | NO   |     | NULL    |                |
| nom            | varchar(30)          | YES  | MUL | NULL    |                |
| commentaires   | text                 | YES  |     | NULL    |                |
| espece_id      | smallint(6) unsigned | NO   | MUL | NULL    |                |
| race_id        | smallint(6) unsigned | YES  | MUL | NULL    |                |
| mere_id        | smallint(6) unsigned | YES  | MUL | NULL    |                |
| pere_id        | smallint(6) unsigned | YES  | MUL | NULL    |                |
| disponible     | tinyint(1)           | YES  |     | 1       |                |
+----------------+----------------------+------+-----+---------+----------------+
```

On retrouve les attributs, les types (domaines). On voit aussi la clé primaire auto-incrémentée.

On voit aussi les liens avec la table espèce et la table race.

Une autre entité est les clients stockés dans la table `Client` :

```
DESCRIBE Client;
+----------------+----------------------+------+-----+---------+----------------+
| Field          | Type                 | Null | Key | Default | Extra          |
+----------------+----------------------+------+-----+---------+----------------+
| id             | smallint(5) unsigned | NO   | PRI | NULL    | auto_increment |
| nom            | varchar(100)         | NO   |     | NULL    |                |
| prenom         | varchar(60)          | NO   |     | NULL    |                |
| adresse        | varchar(200)         | YES  |     | NULL    |                |
| code_postal    | varchar(6)           | YES  |     | NULL    |                |
| ville          | varchar(60)          | YES  |     | NULL    |                |
| pays           | varchar(60)          | YES  |     | NULL    |                |
| email          | varbinary(100)       | YES  | UNI | NULL    |                |
| date_naissance | date                 | YES  |     | NULL    |                |
+----------------+----------------------+------+-----+---------+----------------+
```

Le lien entre les entités animaux et les entités clients est la table `Adoption`.

```
DESCRIBE Adoption;
+------------------+----------------------+------+-----+---------+-------+
| Field            | Type                 | Null | Key | Default | Extra |
+------------------+----------------------+------+-----+---------+-------+
| client_id        | smallint(5) unsigned | NO   | PRI | NULL    |       |
| animal_id        | smallint(5) unsigned | NO   | PRI | NULL    |       |
| date_reservation | date                 | NO   |     | NULL    |       |
| date_adoption    | date                 | YES  |     | NULL    |       |
| prix             | decimal(7,2) unsigned| NO   |     | NULL    |       |
| paye             | tinyint(1)           | NO   |     | 0       |       |
+------------------+----------------------+------+-----+---------+-------+
```

Enfin, on a les tables `Espece` et `Race`, qui donne plus d'information sur les animaux.

```
DESCRIBE Espece;
+-------------+----------------------+------+-----+---------+----------------+
| Field       | Type                 | Null | Key | Default | Extra          |
+-------------+----------------------+------+-----+---------+----------------+
| id          | smallint(6) unsigned | NO   | PRI | NULL    | auto_increment |
| nom_courant | varchar(40)          | NO   |     | NULL    |                |
| nom_latin   | varchar(40)          | NO   | UNI | NULL    |                |
| description | text                 | YES  |     | NULL    |                |
| prix        | decimal(7,2) unsigned| YES  |     | NULL    |                |
+-------------+----------------------+------+-----+---------+----------------+
```

```
DESCRIBE Race;
+-------------+----------------------+------+-----+---------+----------------+
| Field       | Type                 | Null | Key | Default | Extra          |
+-------------+----------------------+------+-----+---------+----------------+
| id          | smallint(6) unsigned | NO   | PRI | NULL    | auto_increment |
| nom         | varchar(40)          | NO   |     | NULL    |                |
| espece_id   | smallint(6) unsigned | NO   | MUL | NULL    |                |
| description | text                 | YES  |     | NULL    |                |
| prix        | decimal(7,2) unsigned| YES  |     | NULL    |                |
+-------------+----------------------+------+-----+---------+----------------+
```

✯ Opérateurs de l'algèbre relationnelle

Pour projeter sur les noms des animaux (ie ne voir que les noms) :

```

```

Pour afficher toute la table :

```

```

On peut voir que l'ordre est arbitraire.
On peut aussi classer par date de naissance :

Sélectionner les trois animaux les plus jeunes :

Moins utile : les trois suivants (ordonnés par date de naissance), avec `OFFSET`

```
SELECT * FROM Animal ORDER BY date_naissance DESC LIMIT 3 OFFSET 3;
```

On peut aussi sélectionner les

La projection ne renvoie pas des valeurs distinctes :

```
SELECT Sexe FROM Animal;
```

renverra ainsi plein de `F`, `M` et de `NULL`.

Si on veut les valeurs possibles, il faut faire :

```
SELECT DISTINCT Sexe FROM Animal;
```

Pour chercher des informations sur les espèce à moins de 150 euros, on fait une restriction suivie d'une projection :

Autre exemple, trouver les noms des animaux nés en 2012 :

Si besoin on peut classer par date de naissance avec `ORDER BY`.

Autre exemple : trouver les personne qui ont adopté et pas payé :

Pour aller un peu plus loin, on peut utiliser des requêtes emboîtées (ou combinées), c'est-à-dire que l'on peut utiliser une requête qui renvoie un nombre dans une condition.

Par exemple :

```
SELECT nom_courant, description, prix
FROM Espece
WHERE prix < ( SELECT AVG(prix) FROM Espece);
```

renvoie le nom courant et la description des espèces moins chère que la moyenne des autres.

⋆ Les jointures

Si par exemple, on veut le nom, le nom de l'espèce de chaque animal cela donne :

Idem avec deux jointures et une restriction pour avoir le nom, le nom de l'espèce, le nom de la race pour les chiens et chats :

Toujours avec deux jointures pour avoir les couples (nom du client, nom de l'animal), avec un renommage pour bien différentier le nom du client et le nom de l'animal :

Autre exemple, jointure d'une table deux fois avec elle-même pour obtenir les parents d'un animal :

```
SELECT Animal.nom AS nomAnimal, Pere.nom AS nomPere, Mere.nom AS nomMere
FROM Animal
JOIN Animal AS Pere ON Animal.pere_id = Pere.id
JOIN Animal AS Mere ON Animal.mere_id = Mere.id
WHERE (Animal.pere_id IS NOT NULL) OR (Animal.mere_id IS NOT NULL);
```

⋆ Agrégation et fonctions d'agrégation

Si on veut le nombre d'animaux :

Le nombre de mâles :

Le nombre de chiens et chat (avec une jointure pour le nom de l'espèce) :

Le total du prix des animaux en boutique (le prix étant donné par l'espèce de l'animal) :

R Avec une structure `IF` présente en Mysql, on pourrait donner le prix en prenant en compte le prix de la race si celle-ci est non `NULL` sinon le prix de l'espèce.

Maintenant si on veut compter le nombre d'animal par sexe :

On veut compter le nombre d'animal par espèce avec une jointure pour avoir le nom de l'espèce :

Avec une agrégation sur deux attributs, on peut compter le nombre de mâles / femelles par espèce :

Le nom et prénom des clients ayant fait plus de deux achats et leur nombre d'adoption :

Ici on filtre après avoir calculé le nombre d'adoption, puisque c'est le critère de filtrage.

Par contre, si on veut compter combien d'animaux par espèce mais sans prendre en compte les chiens et les chats. La bonne méthode est :

puisqu'on efface de la liste dès le début les chats et les chiens.

La mauvaise méthode est :

```
SELECT Espece.nom_courant, count(*)
FROM Animal
JOIN Espece ON Espece.id = Animal.espece_id
GROUP BY Animal.espece_id
HAVING (Espece.nom_courant <> 'Chien') AND  (Espece.nom_courant <> 'Chat');
```

puisqu'on efface simplement deux lignes de la table obtenue par agrégation.

Dernier exemple, en sélectionnant avant l'agrégation et après : on veut le nom et prénom des clients ayant adopté plus de deux chats.

Exemples de requêtes MySQL

Pelletier Sylvain PSI, LMSC

Dans cette fiche on montre les exemples à connaître de requêtes sur une base de données. Pour illustrer ces requêtes, on considère une base de données de taille moyenne concernant un élevage commercial d'animaux.

Pour pouvoir tester les requêtes, nous allons utiliser une base données qui a été déposée en ligne sur le serveur (gratuit) du site `alwaysdata`. Nous allons utiliser l'interface en ligne `phpmyadmin` pour interroger le serveur de base de données.

- Depuis internet, se rendre sur le site : `https://phpmyadmin.alwaysdata.com`
- Rentrer l'identifiant :
- Rentrer le mot de passe :
- Sélectionner la base de données : `psilmsc_test` (onglet de gauche).
- Sur la partie centrale onglet `structure`, se familiariser avec l'interface. en particulier `structure` et `parcourir`.
- Pour écrire une requête cliquer sur l'onglet `SQL`. Cocher conserver la boite de requête puis écrire la requête.
- Il est très pratique d'utiliser un bloc note et de copier / coller du bloc note vers `phpmyadmin`.

Cette base de données fictive provient du cours : Administrez vos bases de données avec MySQL, disponible sur le site de site openclassrooms.

Cette base de données est constituée ainsi :

```
SHOW TABLES;
+------------------+
| Tables_in_testPC |
+------------------+
| Adoption         |
| Animal           |
| Client           |
| Espece           |
| Race             |
+------------------+
```

La table `Animal` contient la liste des animaux dans le magasin :

```
DESCRIBE Animal;
+----------------+----------------------+------+-----+---------+----------------+
| Field          | Type                 | Null | Key | Default | Extra          |
+----------------+----------------------+------+-----+---------+----------------+
| id             | smallint(6) unsigned | NO   | PRI | NULL    | auto_increment |
| sexe           | char(1)              | YES  |     | NULL    |                |
| date_naissance | datetime             | NO   |     | NULL    |                |
| nom            | varchar(30)          | YES  | MUL | NULL    |                |
| commentaires   | text                 | YES  |     | NULL    |                |
| espece_id      | smallint(6) unsigned | NO   | MUL | NULL    |                |
| race_id        | smallint(6) unsigned | YES  | MUL | NULL    |                |
| mere_id        | smallint(6) unsigned | YES  | MUL | NULL    |                |
| pere_id        | smallint(6) unsigned | YES  | MUL | NULL    |                |
| disponible     | tinyint(1)           | YES  |     | 1       |                |
+----------------+----------------------+------+-----+---------+----------------+
```

On retrouve les attributs, les types (domaines). On voit aussi la clé primaire auto-incrémentée.

On voit aussi les liens avec la table espèce et la table race.

Une autre entité est les clients stockés dans la table `Client` :

```
DESCRIBE Client;
+----------------+----------------------+------+-----+---------+----------------+
| Field          | Type                 | Null | Key | Default | Extra          |
+----------------+----------------------+------+-----+---------+----------------+
| id             | smallint(5) unsigned | NO   | PRI | NULL    | auto_increment |
| nom            | varchar(100)         | NO   |     | NULL    |                |
| prenom         | varchar(60)          | NO   |     | NULL    |                |
| adresse        | varchar(200)         | YES  |     | NULL    |                |
| code_postal    | varchar(6)           | YES  |     | NULL    |                |
| ville          | varchar(60)          | YES  |     | NULL    |                |
| pays           | varchar(60)          | YES  |     | NULL    |                |
| email          | varbinary(100)       | YES  | UNI | NULL    |                |
| date_naissance | date                 | YES  |     | NULL    |                |
+----------------+----------------------+------+-----+---------+----------------+
```

Le lien entre les entités animaux et les entités clients est la table `Adoption`.

```
DESCRIBE Adoption;
+------------------+----------------------+------+-----+---------+-------+
| Field            | Type                 | Null | Key | Default | Extra |
+------------------+----------------------+------+-----+---------+-------+
| client_id        | smallint(5) unsigned | NO   | PRI | NULL    |       |
| animal_id        | smallint(5) unsigned | NO   | PRI | NULL    |       |
| date_reservation | date                 | NO   |     | NULL    |       |
| date_adoption    | date                 | YES  |     | NULL    |       |
| prix             | decimal(7,2) unsigned | NO  |     | NULL    |       |
| paye             | tinyint(1)           | NO   |     | 0       |       |
+------------------+----------------------+------+-----+---------+-------+
```

Enfin, on a les tables `Espece` et `Race`, qui donne plus d'information sur les animaux.

```
DESCRIBE Espece;
+-------------+----------------------+------+-----+---------+----------------+
| Field       | Type                 | Null | Key | Default | Extra          |
+-------------+----------------------+------+-----+---------+----------------+
| id          | smallint(6) unsigned | NO   | PRI | NULL    | auto_increment |
| nom_courant | varchar(40)          | NO   |     | NULL    |                |
| nom_latin   | varchar(40)          | NO   | UNI | NULL    |                |
| description | text                 | YES  |     | NULL    |                |
| prix        | decimal(7,2) unsigned | YES |     | NULL    |                |
+-------------+----------------------+------+-----+---------+----------------+
```

```
DESCRIBE Race;
+-------------+----------------------+------+-----+---------+----------------+
| Field       | Type                 | Null | Key | Default | Extra          |
+-------------+----------------------+------+-----+---------+----------------+
| id          | smallint(6) unsigned | NO   | PRI | NULL    | auto_increment |
| nom         | varchar(40)          | NO   |     | NULL    |                |
| espece_id   | smallint(6) unsigned | NO   | MUL | NULL    |                |
| description | text                 | YES  |     | NULL    |                |
| prix        | decimal(7,2) unsigned | YES |     | NULL    |                |
+-------------+----------------------+------+-----+---------+----------------+
```

⋆ Opérateurs de l'algèbre relationnelle

Pour projeter sur les noms des animaux (ie ne voir que les noms) :

```
SELECT nom FROM Animal;
```

Pour afficher toute la table :

```
SELECT * FROM Animal;
```

On peut voir que l'ordre est arbitraire.
On peut aussi classer par date de naissance :

```
SELECT * FROM Animal ORDER BY date_naissance DESC;
```

Sélectionner les trois animaux les plus jeunes :

```
SELECT * FROM Animal ORDER BY date_naissance DESC LIMIT 3;
```

Moins utile : les trois suivants (ordonnés par date de naissance), avec `OFFSET`

```
SELECT * FROM Animal ORDER BY date_naissance DESC LIMIT 3 OFFSET 3;
```

On peut aussi sélectionner les
La projection ne renvoie pas des valeurs distinctes :

```
SELECT Sexe FROM Animal;
```

renverra ainsi plein de `F`, `M` et de `NULL`.
Si on veut les valeurs possibles, il faut faire :

```
SELECT DISTINCT Sexe FROM Animal;
```

Pour chercher des informations sur les espèce à moins de 150 euros, on fait une restriction suivie d'une projection :

```
SELECT nom_courant, description, prix FROM Espece WHERE prix < 150;
```

Autre exemple, trouver les noms des animaux nés en 2012 :

```
SELECT nom, date_naissance FROM Animal WHERE date_naissance > '2012-01-01';
SELECT nom, date_naissance FROM Animal WHERE YEAR(date_naissance) >= 2012;
```

Si besoin on peut classer par date de naissance avec `ORDER BY`.
Autre exemple : trouver les personne qui ont adopté et pas payé :

```
SELECT * FROM Adoption WHERE (date_adoption IS NOT NULL) AND (paye = 0) ;
```

Pour aller un peu plus loin, on peut utiliser des requêtes emboîtées (ou combinées), c'est-à-dire que l'on peut utiliser une requête qui renvoie un nombre dans une condition.
Par exemple :

```
SELECT nom_courant, description, prix
FROM Espece
WHERE prix < ( SELECT AVG(prix) FROM Espece);
```

renvoie le nom courant et la description des espèces moins chère que la moyenne des autres.

⋆ Les jointures

Si par exemple, on veut le nom, le nom de l'espèce de chaque animal cela donne :

```
SELECT Animal.nom, Espece.nom_courant FROM Animal
JOIN Espece
ON Espece.id = Animal.espece_id;
```

Idem avec deux jointures et une restriction pour avoir le nom, le nom de l'espèce, le nom de la race pour les chiens et chats :

```
SELECT Animal.nom, Espece.nom_courant, Race.nom FROM Animal
JOIN Espece
ON Espece.id = Animal.espece_id
JOIN Race
ON Race.id = Animal.race_id
WHERE Espece.nom_courant = 'Chien' OR Espece.nom_courant = 'Chat';
```

Toujours avec deux jointures pour avoir les couples (nom du client, nom de l'animal), avec un renommage pour bien différentier le nom du client et le nom de l'animal :

```
SELECT Client.nom AS nomClient, Animal.nom AS nomAnimal, Adoption.date_adoption
FROM Adoption
JOIN Client ON Client.id = Adoption.client_id
JOIN Animal ON Animal.id = Adoption.animal_id
WHERE Adoption.date_adoption IS NOT NULL;
```

Autre exemple, jointure d'une table deux fois avec elle-même pour obtenir les parents d'un animal :

```
SELECT Animal.nom AS nomAnimal, Pere.nom AS nomPere, Mere.nom AS nomMere
FROM Animal
JOIN Animal AS Pere ON Animal.pere_id = Pere.id
JOIN Animal AS Mere ON Animal.mere_id = Mere.id
WHERE (Animal.pere_id IS NOT NULL) OR (Animal.mere_id IS NOT NULL);
```

⋆ Agrégation et fonctions d'agrégation

Si on veut le nombre d'animaux :

```
SELECT count(*) FROM Animal;
```

Le nombre de mâles :

```
SELECT count(*) FROM Animal WHERE sexe = 'M';
```

Le nombre de chiens et chat (avec une jointure pour le nom de l'espèce) :

```
SELECT count(*) FROM Animal
JOIN Espece
ON Espece.id = Animal.espece_id
WHERE Espece.nom_courant = "Chien" OR Espece.nom_courant = "Chat";
```

Le total du prix des animaux en boutique (le prix étant donné par l'espèce de l'animal) :

```
SELECT sum(Espece.prix) FROM Animal
JOIN Espece
ON Espece.id = Animal.espece_id;
```

R Avec une structure `IF` présente en Mysql, on pourrait donner le prix en prenant en compte le prix de la race si celle-ci est non `NULL` sinon le prix de l'espèce.

Maintenant si on veut compter le nombre d'animal par sexe :

```
SELECT sexe, count(*) FROM Animal GROUP BY sexe;
```

On veut compter le nombre d'animal par espèce avec une jointure pour avoir le nom de l'espèce :

```
SELECT Espece.nom_courant, count(*)
FROM Animal
JOIN Espece ON Espece.id = Animal.espece_id
GROUP BY Animal.espece_id;
```

Avec une agrégation sur deux attributs, on peut compter le nombre de mâles / femelles par espèce :

```
SELECT Animal.sexe, Espece.nom_courant,  COUNT(*)
FROM Animal
JOIN Espece ON Espece.id = Animal.espece_id
GROUP BY Espece.nom_courant, Animal.sexe;
```

Le nom et prénom des clients ayant fait plus de deux achats et leur nombre d'adoption :

```
SELECT Client.nom AS nomClient, Client.prenom AS prenomClient, count(*) AS nbrAdoption
FROM Adoption
JOIN Client ON Adoption.client_id = Client.id
GROUP BY Adoption.client_id
HAVING nbrAdoption > 1;
```

Ici on filtre après avoir calculé le nombre d'adoption, puisque c'est le critère de filtrage.

Par contre, si on veut compter combien d'animaux par espèce mais sans prendre en compte les chiens et les chats. La bonne méthode est :

```
SELECT Espece.nom_courant, count(*)
FROM Animal
JOIN Espece ON Espece.id = Animal.espece_id
WHERE (Espece.nom_courant <> 'Chien') AND  (Espece.nom_courant <> 'Chat')
GROUP BY Animal.espece_id;
```

puisqu'on efface de la liste dès le début les chats et les chiens.

La mauvaise méthode est :

```
SELECT Especе.nom_courant, count(*)
FROM Animal
JOIN Especе ON Especе.id = Animal.especе_id
GROUP BY Animal.especе_id
HAVING (Especе.nom_courant <> 'Chien') AND  (Especе.nom_courant <> 'Chat');
```

puisqu'on efface simplement deux lignes de la table obtenue par agrégation.

Dernier exemple, en sélectionnant avant l'agrégation et après : on veut le nom et prénom des clients ayant adopté plus de deux chats.

```
SELECT Client.nom AS nomClient, Client.prenom AS prenomClient, count(*) AS nbrAdoption
FROM Adoption
JOIN Client ON Adoption.client_id = Client.id
JOIN Animal ON Adoption.animal_id = Animal.id
JOIN Especе ON Especе.id = Animal.especе_id
WHERE Especе.nom_courant = 'Chat'
GROUP BY Adoption.client_id
HAVING nbrAdoption > 1;
```

Algorithme KNN
Algorithme des k-moyennes
Reconnaissance de caractères par apprentissage supervisé
Algorithme des K-means appliqué aux images numériques en couleur

3 — k plus proches voisins et k-moyennes

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*).

I Algorithme KNN

I.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 données.

■ **Exemple I.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 I.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, ie 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. ■

Pour donner un sens à la classe la plus proche, cela suppose *a minima* de choisir une manière de mesurer les écarts entre les objets, ie 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 (nombres réels, vrai/faux, images, etc.).
Le choix de la distance est donc critique pour la classification. Il n'y a pas de bonnes méthodes pour choisir la distance.

I.2 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.

On considère de plus un élément y de E et on veut lui associer une classe.

Comme problème jouet, on considère donc que $E = \mathbb{R}^2$ 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 : $x_{i1}, x_{i2}, \ldots, x_{iK}$.
- Choisir pour classe de y la classe la plus fréquente parmi celles des plus proches voisins : $(x_{i1}, x_{i2}, \ldots, x_{iK})$.

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 parmi les plus fréquentes au nouvel objet y.

I.3 Mise en place de chaque étape

⋆ Mesure des distances

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

```
def distance(y, lX) :
"""
entrée:
lX = liste d'objets (élément de R**2) de longueur N
y = objet (élément de R**2)
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|]
"""

N = len(X)
lD = [0] *N
for k in range(N):
  lD[k] = d(y , X[k])
return lD
```

On aurait aussi pu faire en une ligne :

```
lD = [ d(y , X[k]) for k in range(N) ]
```

✯ 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.

Rappel sur l'algorithme de tri rapide :

```
def tri(L):
    """
    entrée: L = list
              = la liste que l'on souhaite trier.
    sortie: list
              = la liste triée
    """
    if len(L) <= 1 :
        return L
    pivot = L.pop()
    Lpetit = [ x for x in L if x <= pivot]
    Lgrand = [ x for x in L if x > pivot]
    return tri(Lpetit) + [pivot] + tri(Lgrand)
```

R Cette méthode de programmation est loi d'être optimale : on est obligé de recopier la liste plutôt que la trier sur place. L'avantage est qu'il est plus simple de la comprendre avec cette méthode.

De plus, on voit qu'il est inutile pour les KNN de trier toute la liste, ce qui a une complexité de $\underset{N\infty}{O}(N\ln(N))$. Il serait plus judicieux de ne chercher que les K plus petits éléments.

Ces optimisations sont faites en annexe.

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. On crée donc une liste de couple `[indice, distance]` que l'on trie en prenant en compte uniquement la distance (deuxième élément). On sort ensuite la liste des indices des k plus proches voisins.

```
def KNN(D, K):
    """
    entrée: D = 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.

    """
    # on crée une liste distance indice,
    # c'est elle qui sera triée
    listeAtrier = [ [i, D[i] ] for i in range(len(D))]
    listeAtrier = triRapideInd(listeAtrier)

    # la liste est maintenant triée,
    # on extrait les k premiers indices
    LKNN = []
    for i in range(K):
        LKNN.append( listeAtrier[i][0])
    ## autre méthode:
    ## LKNN = [ listeAtrier[i][0] i in range(K) ]

    return LKNN

def triRapideInd(Li):
    """
    entrée: Li = liste du type indice, valeur
    sortie: Li triée en fonction de la valeur
    """
    if len(Li) <= 1 :
        return Li

    pivot = Li.pop()
    Lpetit = [ x  for x in Li if x[1] <= pivot[1] ]
    Lgrand = [ x for x in Li if x[1] > pivot[1]]
    return ( triRapideInd(Lpetit)
               + [pivot]
               + triRapideInd(Lgrand) )
```

✯ Choisir la classe la plus fréquente

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

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

```
def classe(lKNN, lC):
  """
  entrée:
  lKNN = liste de int de longueur K
                = liste des indices des K plus proches voisins.
  lC = liste de int
```

```
    = liste des classes


  sortie:
  cy =
  valeur de la classe la plus fréquente parmi les kNN
  """
  # déterminer les effectifs de chaque classe
  nbrClasse = max(lC)+1 # on peut aussi mettre en entrée
  compteur = [0]*nbrClasse

  for ind in lKNN :
    compteur[ lC[ind] ] += 1


  # détermine la classe la plus fréquente
  cy = 0 # classe la plus fréquente
  emax = compteur[0] # effectif maximum
  for i in range(nbrClasse) :
    if compteur[i] > emax :
      cy = i
      emax = compteur[i]
  return cy
```

I.4 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 a une classe estimée. Ainsi ligne i et colonne j on met le nombre d'éléments qui ont été classifiée 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.

II Algorithme des k-moyennes

II.1 Classification non supervisée

On considère toujours le problème de la classification mais maintenant on ne suppose pas que les données sont déjà classifiées.

On dispose ainsi de N objets que l'on voudrait répartir en k classes. La valeur de k est choisie par l'utilisateur.

On parle de problème de *clustering* puisqu'il s'agit de créer des groupes. On parle aussi d'*apprentissage non supervisée* puisque l'algorithme apprend à créer des groupes sans partir de données (contrairement aux KNN).

Pour notre problème jouet, on considérera toujours que l'on travaille avec des points de $\mathbb{R}^2$ et la distance euclidienne, mais c'est bien sûr une énorme simplification.

II.2 Principe de l'algorithme des k-moyenne

L'algorithme des k-moyennes part d'une classification aléatoire et l'améliore au fur et à mesure.

Chaque classe possède un CENTRE DE GRAVITÉ, qui est le barycentre des points qui font partie de cette classe.

Le principe est d'initialiser arbitrairement les classes en donnant une valeur aux centres de gravité. Puis on attribue à chaque point la classe dont le centre de gravité est le plus proche.

On remet alors à jour les centres de gravité. Puis on attribue de nouveau à chaque point la classe dont le centre de gravité est le plus proche.

On recommence ainsi, jusqu'à convergence ou dépassement d'un nombre d'itération fixé.

La valeur de K est un paramètre critique de l'algorithme. Il n'y a aucune manière de le choisir automatiquement.
De même, le choix de l'initialisation des classes est critique. Ici on a choisi d'initialiser les centres de gravité en utilisant les K premiers points. Ce choix est discutable.
Enfin, il est a priori possible que l'algorithme des K-means ne converge pas. C'est une heuristique.
Quelques résultats théoriques assurent tout de même que les centres de gravité vont plutôt se séparer.

On considère donc N points de $\mathbb{R}^2$ notés $(X_i)_{i\in[\![0,N-1]\!]}$ que l'on veut classifier en k classes.

- On attribue au premier point X_0 la classe 0. Ainsi, X_0 est le centre de gravité de la classe 0.
  Idem, X_1 devient le centre de gravité de la classe 1, etc. Ce qui permet d'initialiser les centre de gravité noté $(G_j)_{j\in[\![0,K-1]\!]}$ et de classifier les K premiers points.
- Pour chaque point X_i, non encore classifié, on va lui attribuer la classe dont le centre de gravité est le plus proche. Pour cela :
  - on calcule les distances $D_{i,j} = d(X_i, G_j)$ pour toutes les valeurs de j,
  - on détermine alors le centre de gravité le plus proche, ie la valeur de j_0 tel que :

$$D_{i,j_0} = \min_{j\in[\![0,K-1]\!]} D_{i,j}$$

  - on associe à X_i la classe j_0 (ie la classe dont le centre de gravité est le plus proche)
- On remet alors à jour les centres de gravité, en donnant pour valeur à G_j le

barycentre des points de classe j.
- On remet alors à jour la classification des points en leur attribuant la classe dont le centre de gravité est le plus proche.
- On recommence les deux étapes précédentes jusqu'à convergence. Pour savoir si l'algorithme converge, on note si l'un des points a changé de classe ou on s'arrête à un nombre fixé d'itérations.

II.3 Mise en place de chaque étape

✯ Prototype de la fonction

La fonction prend en entrée la liste des coordonnées des points, le nombre de classe demandé, et le nombre d'itération maximal.

La sortie est la liste des coordonnées des centre de gravité et la liste des classes de chaque point.

```
def KMeans(lX, K, nbrIterMax):
    """
    entrée: lX = liste de couple de float de longueur n
               = liste des objets
            K  = int
               = nbr de classes demandées.
            nbrIterMax = int
                       = nbr d'iter maximal
    sortie: lCG = liste de couple de float de longueur k
                = liste des coordonnées des centres de gravité
            lC = liste de int de longueur n
               = liste des classes de chacun des points.
    """
```

✯ Initialisation des centres de gravité

Il suffit de donner au centre de gravité la valeur des coordonnées des points :

```
    lCG = [ [lX[i][0], lX[i][1] ] for i in range(K)]
```

✯ Initialisation des classes

On a alors aussi initialisé les premières valeurs de `lC` :

```
    lC = [i for i in range(K)] + [None for i in range(K,n) ]
```

Le reste des points est classifié en fonction du centre de gravité le plus proche :

```
    # initialisation des classes des points restants:
    for i in range(K, n):
        # on calcule toutes les distances entre le point
        # i et les centres de gravité
        lD = [dist( lCG[j], lX[i] ) for j in range(K)]
        # on recherche le min
        ci = indMin(lD)
        # c'est cette classe que l'on donne au point i
        lC[i] = ci
```

La fonction `indMin` détermine l'indice du minimum est bien connue :

```
def indMin(L):
    """
    entrée: L = list de float
    sortie indMin = indice du minimum
    """
    indMin = 0
    valMin = L[0]
    for i in range(len(L)):
        if L[i] < valMin:
            valMin = L[i]
            indMin = i
    return indMin
```

⋆ Boucle principale

On utilise une variable booléenne qui permet de déterminer si il y a eu changement et un compteur pour le nombre d'itération. La boucle principale est ainsi une boucle :

```
    nbrIter = 0
    chgt = True # = False si pas de chgt durant un tour
    while chgt and nbrIter < nbrIterMax :
        chgt = False
```

La variable `chgt` est mise à `False` dès le début de la boule et ne passe à `True` que si un point change de classe.

⋆ Mise à jour des centres de gravité

Pour mettre à jour les centre de gravité, c'est une simple adaptation de l'algorithme de moyenne.

```
moyX = [0 for i in range(K)]
moyY = [0 for i in range(K)]
eff = [0 for i in range(K)] # effectif de chaque classe
for i in range(n):
    x,y = lX[i]
    c = lC[i]
    moyX[c] += x
    moyY[c] += y
    eff[c] += 1
lCG = [ [moyX[c] /eff[c], moyY[c] / eff[c] ]
                                    for c in range(K) ]
```

REM : on admet que l'effectif d'une classe ne peut être nul.

⋆ Mise à jour de la classification

C'est la même technique que celle vue pour l'initialisation. La seule différence est que l'on note si il y a eu changement dans la classification d'un point.

```
        for i in range(n):
            # on calcule toutes les distances entre le point
            # i et les centres de gravité
            lD = [dist( lCG[j], lX[i] ) for j in range(K)]
            # on recherche le min
```

```
            ci = indMin(lD)
            # c'est cette classe que l'on donne au point i
            if lC[i] != ci :
                chgt = True
                lC[i] = ci
```

Annexe programmes complets KNN et KMeans

Pelletier Sylvain PSI, LMSC

Ces fichiers permettent de tester les algorithmes sur des données simulées aléatoirement et d'illustrer les algorithmes par des graphiques.

★ **KNN**

```
from pylab import rand, plot, show, sqrt, Circle, subplots, axis

# paramètre qui donne le dispersement des points
# pour le tirage aléatoire
SIGMA = 0.2

# liste des couleurs pour représenter les classes
# NB: doir être plus grande que le nombre
# de classes

LCOULEUR = ["blue", "yellow", "red", "green", "orange", "gray", "purple", "brown", "pink", "oli

def creeDonnee( n, t):
    """
    entrée: n = nbr de points
            t = nbr de classe
    sortie: lX = liste de couple de float de longueur n
               = liste des objets
               = liste des coordonnées des points
            lC = liste de int
               = liste des classes

    crée une liste aléatoire d'objets
    """


    # crée une liste de centre pour chaque classe
    lCentre = [ [rand(), rand()] for i in range(t) ]

    lX = []
    lC = []
    for i in range(n):
        # on choisit aléatoirement une classe à l'objet
        classeX = int(rand()*t)
        # coordonnées aléatoire
        xC, yC = lCentre[ classeX]
        x = xC+ rand()*SIGMA
        y = yC+ rand()*SIGMA

        lC.append( classeX)
        lX.append( [x,y])
    return lX, lC

def representeDonnee(lX, lC):
    """
    entrée: lX = liste de couple de float de longueur n
               = liste des objets
               = liste des coordonnées des points
            lC = liste de int
               = liste des classes
    sortie: rien affichage graphique des objets
    """
    assert len(lX) == len(lC)

```

```
    for i in range(len(lX)):
        x,y = lX[i]
        c = LCOULEUR[ lC[i]]
        plot(x,y, "*", color = c)
#     show(block = False)

def dist(A,B):
    """
    entrée: A,B = couple de float
                = coordonnées de deux points A et B
    sortie: float
            distance AB
    """
    xA, yA = A
    xB, yB = B
    return sqrt( (xA - xB)**2 + (yA - yB)**2 )

def listedesdistances(lX , y):
    """
    entrée: lX = liste de couple de float de longueur n
               = liste des objets
               = liste des coordonnées des points
            y  = couple de float
               = coordonnées du point à placer.
    sortie: D = liste de float
              = liste des distances: D[i]
                est la distance entre le ieme point et y
     """
    D = []
    for i in range(len(lX)):
        D.append( dist(lX[i], y) )
    return D

def KNN(D, K):
    """
    entrée: D = 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.

    """
    # on crée une liste distance indice, c'est elle qui
    # sera triée
    listeAtrier = [ [i, D[i] ] for i in range(len(D))]
    listeAtrier = triRapideInd(listeAtrier)

    LKNN = []
    for i in range(K):
        LKNN.append( listeAtrier[i][0])
    return LKNN









def triRapideInd(Li):
    """
    entrée: Li = liste du type indice, valeur
    sortie: Li triée en fonction de la valeur
    """
```

```
    if len(Li) <= 1 :
        return Li

    pivot = Li.pop()
    Lpetit = [ x  for x in Li if x[1] <= pivot[1] ]
    Lgrand = [ x for x in Li if x[1] > pivot[1]]
    return triRapideInd(Lpetit) + [pivot] + triRapideInd(Lgrand)

def classe(lKNN, lC):
  """
  entrée:
  lKNN = liste de int de longueur K
                = liste des indices des K plus proches voisins.
  lC = liste de int
     = liste des classes


  sortie:
  cy =
  valeur de la classe la plus fréquente parmi les kNN
  """
  # déterminer les effectifs de chaque classe
  nbrClasse = max(lC)+1 # on peut aussi mettre en entrée
  compteur = [0]*nbrClasse

  for ind in lKNN :
    compteur[ lC[ind] ] += 1


  # détermine la classe la plus fréquente
  cy = 0 # classe la plus fréquente
  emax = compteur[0] # effectif maximum
  for i in range(nbrClasse) :
    if compteur[i] > emax :
      cy = i
      emax = compteur[i]
  return cy



print("début du programme des k plus proches voisins")

fig, ax = subplots()

print("simulation de données aléatoires")
lX, lC = creeDonnee( 200, 8)
representeDonnee(lX , lC)
print("liste des points dans la base:")
for i in range(len (lX)):
    print("i:", i, "\t coordonnées:", lX[i][0], ",",lX[i][1], " \t classe: ", lC[i])



y = rand(), rand()
print("point à placer:", y[0], ",", y[1])
plot(y[0],y[1], ".", color="black")

D = listedesdistances(lX , y)



LKNN = KNN(D, 8)
print("voici les points les plus proches")
for ind in LKNN :
```

```
    print("point d'indice", ind,  "\t coordonnées:", lX[ind][0], ", ", lX[ind][1], "\t classe:"
184
cy = classe(LKNN, lC)
186 print("classe du point y", cy, "il est colorié en ", LCOULEUR[cy])

188 # dessin du cercle
distmax = dist(y, lX[ LKNN[-1]])
190


192 cercle = Circle(y, distmax, fill = False)
ax.add_patch(cercle)
194 axis("equal")
show()
```

⋆ Optimisation KNN

La première optimisation consiste à ne pas trier toute la liste des couples `indice, valeur` mais uniquement les K premiers éléments.

Ainsi :

- si la liste à trier contient moins de K éléments, on s'arrête,
- si `LPetit` est de taille supérieure à K, on ne trie pas `LGrand`, on se relance sur `LPetit`.
- Si `LPetit` est de taille inférieure à K, inutile de la trier, il faut chercher l'élément `K - len(LPetit) -1` dans `LGrand` puis renvoyer `LPetit`, suivi du pivot, suivi du résultat ainsi obtenu.

La fonction devient alors :

```
def rechercheKInd(Li, K):
2     """
      entrée: Li = liste du type indice, valeur
4             K = int
      sortie: liste de longueur K contenant des couples indices / valeurs avec les K plus petites
6     """
      if len(Li) <= K :
8         return Li

10    pivot = Li.pop()

12    Lpetit = [ x  for x in Li if x[1] <= pivot[1] ]
      Lgrand = [ x for x in Li if x[1] > pivot[1]]
```

Ainsi programmé, l'algorithme est en complexité $O\left(K\ln(N)\right)$.

Une deuxième optimisation consiste à classer la liste sur place, sans utiliser de liste secondaire. Il faut alors éviter l'utilisation de `lPetit` et `lGrand` et déplacer des valeurs dans la liste. On propose d'utiliser des indices pour la position du pivot. voici un code possible :

```
def rechercheKInd(Li, K):
2     """
      entrée: Li = liste du type indice, valeur
4             K = int
      sortie: liste de longueur K contenant des couples indices / valeurs avec les K plus petites
6     """
      if len(Li) <= K :
8         return Li

10    placePivot = 0
      pivot = Li[0]
12
      for i in range(1,len(Li)):
14        # placement du ieme terme.
          if Li[i][1] <= pivot[1]:
16            # il faut placer l'élément i avant  le pivot
              # on permute alors les trois éléments:
18            # à la place du pivot, on met le ieme terme
              # le pivot est temporairement stocké en position i
20            Li[placePivot], Li[i] = Li[i],Li[placePivot]
```

```
            # on change la place du pivot:
            placePivot += 1
            # on remet le pivot en place
            Li[placePivot], Li[i] = Li[i],Li[placePivot]
        assert pivot == Li[placePivot] # verif


    # arrivé ici les placePivot +1 éléments sont classés dans le bon ordre.
    # on fait un appel récursif
    if placePivot +1 >= K :
        return rechercheKInd(Li[: placePivot+1], K)
    else :
        return Li[:placePivot+1] + rechercheKInd(Li[placePivot+1:], K - (placePivot+1))
```

⋆ KMeans

```
from pylab import rand, plot, show, sqrt, Circle, subplots, axis, figure, title

# paramètre qui donne le dispersement des points
# pour le tirage aléatoire
SIGMA = 0.2

# liste des couleurs pour représenter les classes
# NB: doir être plus grande que le nombre
# de classes

LCOULEUR = ["blue", "yellow", "red", "green", "orange", "gray", "purple", "brown", "pink", "oli

def creeDonnee( n, t):
    """
    entrée: n = nbr de points
            t = nbr de classe
    sortie: lX = liste de couple de float de longueur n
               = liste des objets
               = liste des coordonnées des points
            lC = liste de int
               = liste des classes

    crée une liste aléatoire d'objets
    """


    # crée une liste de centre pour chaque classe
    lCentre = [ [rand(), rand()] for i in range(t) ]

    lX = []
    for i in range(n):
        # on choisit aléatoirement une classe à l'objet
        classeX = int(rand()*t)
        # coordonnées aléatoire
        xC, yC = lCentre[ classeX]
        x = xC+ rand()*SIGMA
        y = yC+ rand()*SIGMA

        lX.append( [x,y])
    return lX

def representePoints(lX):
    """
    entrée: lX = liste de couple de float de longueur n
               = liste des objets
               = liste des coordonnées des points
    sortie: rien affichage graphique des objets
    """
```

```
    for i in range(len(lX)):
        x,y = lX[i]
        plot(x,y, "*", color = "black")



def representePointsClasse(lX, lC, lCG):
    """
    entrée: lX = liste de couple de float de longueur n
               = liste des objets
               = liste des coordonnées des points
            lC = liste de int ou liste vide
               = liste des classes de chaque point
            lCG = liste de couple de float de longueur k
                = liste des centre de gravité
    sortie: rien affichage graphique des objets
    """
    for i in range(len(lX)):
        x,y = lX[i]
        c = LCOULEUR[ lC[i]]
        plot(x,y, "*", color = c)
    for i in range(len(lCG)) :
        x,y = lCG[i]
        c = LCOULEUR[i]
        plot(x,y, "o", color = c)


def dist(A,B):
    """
    entrée: A,B = couple de float
                = coordonnées de deux points A et B
    sortie: float
            distance AB
    """
    xA, yA = A
    xB, yB = B
    return sqrt( (xA - xB)**2 + (yA - yB)**2 )


def KMeans(lX, K, nbrIterMax):
    """
    entrée: lX = liste de couple de float de longueur n
               = liste des objets
            K  = int
               = nbr de classes demandées.
            nbrIterMax = int
                       = nbr d'iter maximal
    sortie: lCG = liste de couple de float de longueur k
                = liste des coordonnées des centres de gravité
            lC = liste de int de longueur n
               = liste des classes de chacun des points.
    """
    n = len(lX)
    # initialisation aléatoire des centres de gravité
    # on initialise avec les k premiers points
    lCG = [ [lX[i][0], lX[i][1] ] for i in range(K)]
    lC = [i for i in range(K)] + [None for i in range(K,n) ]

    # initialisation des classes des points restants:
    for i in range(K, n):
        # on calcule toutes les distances entre le point
        # i et les centres de gravité
        lD = [dist( lCG[j], lX[i] ) for j in range(K)]
        # on recherche le min
```

```
113         ci = indMin(lD)
            # c'est cette classe que l'on donne au point i
115         lC[i] = ci
        figure()
117     representePointsClasse(lX, lC, lCG)
        title("initialisation des classes")
119     show(block = False)


121


123     # boucle principale
        nbrIter = 0
125     chgt = True # = False si pas de chgt durant un tour
        while chgt and nbrIter < nbrIterMax :
127         chgt = False
            print("KMeans iter:", nbrIter)
129
            # on calcule les centre de gravité
131         # simple calcul de moyenne
            moyX = [0 for i in range(K)]
133         moyY = [0 for i in range(K)]
            eff = [0 for i in range(K)] # effectif de chaque classe
135         for i in range(n):
                x,y = lX[i]
137             c = lC[i]
                moyX[c] += x
139             moyY[c] += y
                eff[c] += 1
141         lCG = [ [moyX[c] /eff[c], moyY[c] / eff[c] ] for c in range(K) ]
            # NB: l'analyse théorique assure que eff[c] > 0 pour chaque c
143


145         # on attribue à chaque points la classe
            # dont le centre de gravité est le plus proche
147         for i in range(n):
                # on calcule toutes les distances entre le point
149             # i et les centres de gravité
                lD = [dist( lCG[j], lX[i] ) for j in range(K)]
151             # on recherche le min
                ci = indMin(lD)
153             # c'est cette classe que l'on donne au point i
                if lC[i] != ci :
155                 chgt = True
                    lC[i] = ci
157
            nbrIter += 1
159         figure()
            representePointsClasse(lX, lC, lCG)
161         title("iter " + str(nbrIter))
            show(block = False)
163     if not chgt :
            print("l'algorithme a convergé en ", nbrIter, "itérations")
165     return lC, lCG

167 def indMin(L):
        """
169     entrée: L = list de float
        sortie indMin = indice du minimum
171     """
        indMin = 0
173     valMin = L[0]
        for i in range(len(L)):
175         if L[i] < valMin:
                valMin = L[i]
```

```
177                 indMin = i
        return indMin
179
print("début du programme des K moyennes")
181
183 print("simulation de données aléatoires")
lX = creeDonnee( 200, 10)
185 print("liste des points dans la base:")
for i in range(len (lX)):
187     print("i:", i, "\t coordonnées:", lX[i][0], ",",lX[i][1])
figure()
189 representePoints(lX)
title("points non classés")
191 show(block = False)

193 lC, lCG = KMeans(lX, 5, 100)
for i in range(len(lCG)) :
195     print("cluster ", i, "centre:", lCG[i][0], ",",lCG[i][0] , "couleur:", LCOULEUR[i] )
figure()
197 representePointsClasse(lX, lC, lCG)
title("état final")
199 show(block = False)
```

Reconnaissance de caractères par apprentissage supervisé

Pelletier Sylvain PSI, LMSC

Dans ce TD, on considère le problème suivant : à partir d'une base d'apprentissage, on souhaite reconnaître automatiquement des chiffres manuscrits sur une image.

Précisément, on dispose d'une base d'images en noir et blanc, toutes de taille 28×28. Chaque image représente un chiffre écrit à la main, on sait quel chiffre a été écrit. Il s'agit d'un extrait de la base MNIST, qui est un standard pour tester ce type d'algorithme. Avec cette base, on souhaite classifier d'autres images de chiffres manuscrits, de même taille, pour pouvoir reconnaître automatiquement le chiffre écrit à la main. On dispose donc aussi d'une base de test. Ces tests sont aussi classifiés, ce qui va permettre d'évaluer la qualité de notre algorithme.

L'algorithme que l'on va utiliser est celui des KNN (K plus proches voisins) que l'on a étudié en cours.

On rappelle ici le principe :

- on considère un ensemble E (ici c'est donc les images en niveau de gris de taille 28×28), on munit cet ensemble d'une distance d.
  On considère aussi un ensemble C de classe (ici les classes sont les entiers de $[\![0,9]\!]$).
- On dispose de N éléments de $E \times C$. C'est le N images classifiées pour lesquelles on connaît le chiffre qui est écrit. On les note $(x_j, c_j)_{j \in [\![1,N]\!]}$.
- On considère un élément $y \in E$ que l'on souhaite classifier : c'est une image (toujours en niveau de gris de taille 28×28) sur laquelle on ne sait pas quel chiffre est écrit.
- L'algorithme consiste à
  - mesurer les distances $d(y, x_j)$ pour $j \in [\![1,N]\!]$, il faut donc décider d'une distance entre images.
  - déterminer les k plus proches voisins de y. Les k éléments les plus proches de y sont notés : $x_{i1}, x_{i2}, \ldots, x_{ik}$. Il faut donc classer les images de la base en fonction de la distance avec y.
  - Choisir pour classe de y la classe la plus fréquente parmi celles correspondantes à $(x_{i1}, x_{i2}, \ldots, x_{ik})$, ie $(c_{i1}, c_{i2}, \ldots, c_{ik})$.

⋆ Structure des données

Dans ce td on a donc besoin d'une base de données images.

Avec les fichiers fournis, on dispose des instructions :

```
from dataPetite import data, test
from dataGrande import data, test
from dataTotale import data, test
```

Il faut décommenter une de ces lignes et une seule pour importer les données. Bien entendu, on commence par les petites données pour bien tester, et on n'utilise les grandes données que pour des tests plus poussés (pour tester les performances de notre algorithme).

Les données totales ne sont accessibles que si vous avez installé le module `keras` de Python (voir en fin de td pour les instructions). Elles ne sont pas nécessaires pour faire le td.

Les objets `data` et `test` sont des dictionnaires `cle:valeur`, construits ainsi :

cle est le nom de l'image. C'est une chaîne de caractères de la forme `i_j` où $i \in [\![0,9]\!]$ donne le chiffre représenté, et j est le numéro de l'image dans la base.
La première lettre de la clé indique ainsi quel chiffre est écrit à la main.

valeur est un `array` de `float` 28×28 représentant donc l'image du chiffre i manuscrit.
Une image en niveau de gris est simplement un tableau de nombres flottants. Un chiffre élevé représente un pixel clair, tandis qu'un chiffre faible représente un pixel sombre.

Par exemple, pour afficher la liste des noms des images et les chiffres inscrits, on peut utiliser les instructions :

```
for cle in data:
  print("l'image "+ cle + " représente le chiffre " + cle[0])
```

Le dictionnaire `test` contient la même structure avec cette fois les images à tester. On connaît donc le chiffre qui est inscrit sur les tests, ce qui permet d'évaluer l'efficacité de notre algorithme. On pourra ainsi comparer le chiffre reconnu par

l'algorithme (classification estimée) dans une image de test par rapport au vrai chiffre inscrit (classification certaine) qui est toujours la première lettre de la clé.

⋆ Distance entre images

La première étape est donc de définir une distance entre images. Les images sont stockées sous la forme d'un tableau (structure `array` de `numpy`) de nombres flottants. Chaque case (i,j) du tableau / image `A` est un **pixel** contenant donc un `float`. On a $\texttt{A[i,j]} \in [0,255]$.

On cherche à définir une distance entre ces images, qui sont toutes de la même taille.

Une idée simple consiste à faire la somme des différences entre chaque pixel. On définit donc :

$$d(\texttt{A},\texttt{B}) = \sum_{(i,j)\in[\![0,27]\!]^2} |\texttt{A[i,j]} - \texttt{B[i,j]}|$$

C'est bien une distance entre l'image `A` et l'image `B`.

On écrit donc une fonction :

```
def dist(A,B):
    """
    entrée: A,B = array de même taille
                = deux images représentant des chiffres
    sortie: float
            distance AB ie somme des valeurs abs( A[i,j] - B[i,j])
    """
```

Puis une fonction :

```
def listedesdistances(data , img):
    """
    entrée: data =  dictionnaire de array
               = liste des images des chiffres déjà classées
            img  = array
               = image du chiffre à déterminer
    sortie: D = liste de couples de la forme [dist, nom]
                de la même taille que data
              = liste des distances (dist) entre img et data[nom]
    """
```

⋆ Tri fusion

L'étape suivante consiste à utiliser cette liste de distances pour trier les images.

On souhaite donc trier la liste D en fonction de la première valeur (la distance) par ordre croissant.

Pour changer un peu de l'algorithme vu en cours, on se propose d'adapter l'algorithme de tri fusion vu en première année.

Pour rappel l'algorithme de tri fusion consiste à :

- écrire une fonction itérative qui fusionne deux listes (de longueur quelconque) `L1`, `L2` déjà triées en créant une nouvelle liste `L`.

  Le principe est que tant qu'aucune des deux listes n'est vide :
  - on compare les deux premiers éléments.
  - Si `L1[0]> L2[0]` on enlève le premier élément de `L2`, on l'ajoute à `L`,
  - Sinon, on enlève le premier élément de `L1` et on l'ajoute à `L`,

  Lorsqu'une des deux listes est vide, on ajoute l'autre à la fin de `L`.

  Indication : on pourra utiliser les instructions :

  ```
  elmt = L1.pop(0) # enlève le premier élément d'une liste
  L.extend(L1) # ajoute la liste L1 à la fin de la liste L
  ```

  REM : il existe une version itérative de cette fonction. On ne va pas l'utiliser ici car sinon la pile d'appel peut dépasser la taille maximale autorisée en Python.
- écrire une fonction récursive `tri` qui trie une liste quelconque `L` :
  - si `L` est vide ou ne contient qu'un élément, on la renvoie,
  - sinon on la découpe en deux listes `L1`, `L2` :

    ```
    m = len(L)//2
    L1 = L[0:m]
    L2 = L[m:]
    ```

on trie L1 et L2 (par un appel récursif) et on les fusionne.

Il faut donc adapter cet algorithme au cas où on trie des couples en fonction du premier élément. On écrira donc deux fonctions. La première fusionne deux listes

```
def fusion(L1,L2) :
    """
    entrée: L1 et L2 = deux listes de couples  [dist, nom]
                  déjà triées en fontion de dist
                  par ordre croissant
    sortie: L = liste de couples [dist, nom]  triée en fonction de dist
                  par ordre croissant
             = fusion des listes L1 et L2
    """
```

La seconde effectue le tri proprement dit :

```
def triFusion(L):
    """
    entrée: L = liste de couples du type [dist, nom]
    sortie: L triée en fonction de dist
    """
```

✯ Classement par les K plus proches voisins

Une fois que l'on peut trier la liste des distances, on va utiliser ce résultat pour déterminer les k plus proches voisins d'une image. Cette fonction prend en entrée, la liste des couples `[dist, nom]` (ie la sortie de la fonction `listedesdistances`) et détermine la liste des noms des k images les plus proches.

Pour cela, on utilise donc l'algorithme de tri vu à l'étape précédente, pour trier la liste `D`. Puis on extrait de D la liste des k premiers noms, ie la deuxième composante des k premiers couple de D.

On écrit donc une fonction de prototype :

```
def KNN(D, K):
    """
    entrée: D = liste de couples de la forme [dist, nom]
              = liste des distances (dist) entre img et data[nom]
            K = int = nombre de voisins à prendre en compte
    sortie: lKNN = liste de str de longueur K
                 = liste des noms  des K plus proches images  voisines

    """
```

Avec cette liste, il faut déterminer la classe la plus fréquente à partir de cette liste de noms. Pour cela, il faut utiliser des compteurs, et on propose ici d'utiliser des compteurs sous la forme d'un dictionnaire :

- On crée un dictionnaire vide `compteur`.
- on parcourt la liste `lKNN`, et on extrait le chiffre correspondant à la clé par : `c = nom[0]`.
- si la clé `c` n'existe pas dans le dictionnaire `compteur`, on la crée et l'initialise à 1, si elle existe déjà, on l'incrémente de 1.

Ensuite, il n'y a plus qu'à parcourir `compteur` pour faire une recherche de la clé qui a la valeur maximale. C'est cette valeur qui donne le chiffre lu dans l'image de test.

Le prototype de la fonction est :

```
def classe(lKNN):
    """
    entrée: lKNN = liste de str de longueur K
         = liste des noms  des K plus proches images  voisines
    sortie: cPlusfreq = chiffre le plus fréquent parmi les images voisines
    """
```

✯ Création de la matrice de confusion

Une fois que vous avez bien testé votre algorithme sur une unique image (et que l'on a enlevé tous les `print`) de vérification), la première étape consiste à réorganiser le code pour écrire une fonction qui regroupe tout ce qui a été fait.

Cette fonction a pour prototype :

```
def litChiffre(img, data, K):
    """
    entrée: img = array 28x28 de float
                = image à classifier
            data = dictionnaire
                 = liste des images des chiffres déjà classées
            K = int = nombre de voisins à prendre en compte
     sortie: c = chiffre lu dans img par l'algorithme.
    """
```

On enlève alors les instructions `print` inutiles dans le fichier.

Pour tester l'algorithme, on va l'utiliser sur toute la base de test et comparer les résultats obtenus avec la vraie valeur que l'on connaît.

On appelle la **classe certaine** d'une image la "vraie classe" connue dans la base de test et la **classe estimée** la classe obtenue par l'algorithme.

On va alors créer la **matrice de confusion**, c'est un tableau `T` de taille 10×10 (autant que de classes), tel que : `T[i,j]` contient le nombre d'images de tests dont la classe certaine est i et la classe estimée est j.

Idéalement, le tableau T ne contient que des valeurs sur la diagonale.

Il faut donc créer une fonction :

```
def matConfusion(test, data):
    """
    entrée: test = dictionnaire
                 = liste des images à classer
            data = dictionnaire
                 = liste des images des chiffres déjà classées
   sortie: T = array numpry de int
             = matrice de confusion
    """
```

Ensuite, on teste différents paramètres (valeur de K, mais aussi choix de la distance) pour obtenir la matrice de confusion la meilleure possible (celle pour laquelle il y a le moins de valeurs en dehors de la diagonale).

⋆ Test sur la base MNIST

La base MNIST est un standard de test pour la reconnaissance de chiffre (voir la page wikipedia associée).

Elle contient : 60000 images d'apprentissage et 10000 de test. Ces images sont recentrées et mises à la même échelle.

Pour l'utiliser, il est nécessaire d'inclure le module Python `keras`. si vous avez installé le gestionnaire de module en Python pip, il suffit d'utiliser les commandes :

```
pip3 install tensorflow
pip3 install Keras
```

Connecté à internet, le téléchargement et l'installation du module est alors automatique. On peut alors tester sur la base complète.

Attention au temps de calcul, ne pas lancer sur tous les tests de la base.

Algorithme des K-means appliqué aux images numériques en couleur

Pelletier Sylvain PSI, LMSC

Ce TD est fortement inspiré de celui de Mickaël Péchaud `http://mpechaud.fr/`

Dans ce TD, on considère le problème suivant : à partir d'une image numérique couleur, on cherche à réduire le nombre de couleurs. L'idée est de procéder comme un peintre, qui choisirait sa palette de couleur en fonction de l'image et qui ne pourrait utiliser que ces couleurs.

Pour cela, on va utiliser l'algorithme des **K-means** : dans l'image originale, chaque pixel a une couleur qui lui est propre, on va considérer chaque pixel comme une donnée dans l'espace (R,G,B) des couleurs et on va chercher à regrouper toutes ces couleurs en K groupes.

L'image dite « aplatie » est alors obtenue en remplaçant chaque couleur par celle du barycentre du groupe, cette image n'utilise donc que K couleurs, mais ces couleurs ont été choisies selon l'image.

⋆ Représentation des images numériques en couleur

Une **image numérique couleur** correspond à la donnée d'un tableau contenant des triplet d'entiers. Chaque case du tableau (appelée **pixel**) contient les **trois composantes chromatiques** d'un point de l'image : le rouge, le vert et le bleu.

Plus précisément, en python l'image de taille $n \times m$ sera représenté sous la forme d'un `3d-array` `img` de taille $n \times m \times 3$.

- La valeur de `img[i,j,0]` est le niveau de rouge du pixel `(i,j)`,
- la valeur de `img[i,j,1]` le niveau de vert,
- et celle de `img[i,j,2]` le niveau de bleu.

On peut ainsi avoir accès et modifier la couleur du pixel `(i,j)` pour $i \in [\![0, n-1]\!]$ et $j \in [\![0, m-1]\!]$.

R Souvent, `img` est en fait un `3d-array` de taille $n \times m \times 4$. La quatrième composante chromatique est **la transparence** que l'on n'utilisera pas ici.

Selon la manière dont l'image est codée et selon comment elle est lue par le logiciel, cet `array` contiendra :

- des entiers de $[\![0, 255]\!]$ codés sur $8bits$, c'est le type `unit8`.
- des réels de $[0, 1[$.

Dans la suite, on prend la convention que les valeur sont codées sous la forme de `uint8`. Si ce n'est pas le cas, il est facile d'adapter.

! Il y a un piège lorsqu'on manipule des `uint8` : si on fait la somme de deux entiers codés en `uint8`, on peut obtenir un entier supérieur à 255 et donc un dépassement de capacité ! Il faut les convertir en entier (avec `int`) ou en flottants (avec `float`).

Lorsqu'on manipule les images, il est important de savoir dans lequel de ces cas on s'est placé. Pour vérifier, le plus simple est d'afficher la valeur et le type du pixel central comme cela est fait sur le script fourni.

Les différentes couleurs sont obtenues en donnant des valeur aux composantes (R,G,B) niveau de rouge, vert et bleu respectivement.

On obtient ainsi les différentes couleurs :

Le blanc corresponds à $(R,G,B) = (255,255,255)$,
le noir à $(R,G,B) = (0,0,0)$,
le bleu à $(R,G,B) = (0,0,255)$,
le gris moyen à $(R,G,B) = (128,128,128)$,
le jaune à $(R,G,B) = (255,255,0)$
le violet à $(R,G,B) = (255,0,255)$ etc.

En particulier, la luminosité d'une pixel corresponds à la somme des trois composantes. Plus le pixel est clair plus cette somme est importante. En mélangeant les différents niveaux, on a $256^3 = 16777216$ couleurs différentes, ce qui corresponds à la résolution des écrans standards.

⋆ Algorithme des K-means appliqué à la recherche de palettes

Dans ce TP, on considère la couleur de chaque pixel comme un point dans l'espace (R,G,B).

On dispose donc de $N = n \times m$ données. On va traiter ces données avec l'algorithme de K-means en les regroupant en K groupes.

NB : On ne s'intéresse pas à la position des pixels dans l'image, on regarde juste leur couleur.

Rappel sur le principe de l'algorithme de K-means

- On considère N points dans un espace vectoriel normé E, on cherche à les répartir en K classes. Le choix de la valeur K est critique et laissé à l'utilisateur.
- On initialise les K classes aléatoirement. On dispose ainsi du barycentre de chaque classe. Le choix de cette initialisation est critique et laissé à l'utilisateur.
- On répète alors les deux opérations suivantes jusqu'à convergence :
  - associer à chaque points la classe du barycentre le plus proche,
  - recalculer les barycentres.

Dans notre algorithme, on essaiera $K = 7, 9$ ou 11.

⋆ Initialisation

Pour initialiser les classes aléatoirement, on propose de tirer avec `rand` un point au hasard dans l'espace (R,G,B). Par exemple, le code :

```
r = int(rand()*256)
```

permet de calculer une valeur au hasard.

Écrire alors une fonction :

```
def initialisation(K):
    """
    entrée: K = int
              = nbr de couleurs dans la palette
    sortie: lBary = list de triplet de int
                  = coordonnées aléatoires des K barycentres
    """
```

Pour tester et vérifier, on pourra utiliser la fonction `affichePalette` fournie.

⋆ Mise à jour des classes

Pour l'algorithme des K-means, il faut décider d'une distance entre les éléments, ici donc entre les couleurs. Le choix de cette distance est critique et laissée à l'utilisateur. Si on change de distance, on obtient des résultats très différents.

Dans ce td, on choisir la distance suivante :

```
def distCouleur(c1, c2):
    """
    entrée: c1 , c2 = triplet de uint8
                    = couleurs
    sortie: d = float
              = distance entre les couleurs
    """
    r1, g1, b1 = c1
    r2, g2, b2 = c2
    # conversion flottant:
    fr1, fg1, fb1 = float(r1), float(g1), float(b1)
    fr2, fg2, fb2 = float(r2), float(g2), float(b2)

    return sqrt( (fr1-fr2)**2 + (fg1-fg2)**2 + (fb1-fb2)**2 )
```

Dans un deuxième temps, on pourra changer cette fonction.

À partir de cette fonction distance, on met alors à jour les classes en utilisant la liste des barycentres. Écrire une fonction d'entête :

```
def classe(img, lBary):
    """
    entrée: img = array 3-d (n,m,c)
                = image couleur
```

```
            lBary = list de triplet de int
                  = coordonnées des K barycentres
    sortie: imgc = array 2d de taille (n,m)
                 = contenant des int
                 = imgc[i,j] est un int de [|0, K-1 |]
                   donnant la classe du pixel (i,j)
    """
```

Pour cela, on calcule pour chaque pixel (i, j), la liste des distance entre la couleur de ce pixel et chacun des K barycentres. Cela donne une liste de flottants de taille K, puis on cherche l'indice du minimum de cette liste. C'est le barycentre le plus proche de la couleur du pixel (i, j).

On pourra ainsi utiliser une fonction annexe qui calcule l'indice du minimum d'une liste.

```
def indMin(L):
    """
    entrée: L = list de float
    sortie indMin = indice du minimum
    """
```

La fonction `aplat` (fournie) permet de construire une image à partir des classes et de la liste des barycentres.

```
def aplat(imgc, lBary):
    """
    entrée: imgc = array 2d de même taille que img
                 = imgc[i,j] est un int de [|0, K-1 |]
                     donnant la classe du pixel (i,j)
            lBary = list de triplet de int
                  = coordonnées des K barycentres
    sortie: imgaplat = array 3d de même taille que img
                     = image couleur obtenue en mettant la couleur du barycentre
                       du groupe de chaque pixel
    """
    n,m = shape(imgc)
    imgaplat = zeros((n,m,3), int )
    for i in range(n):
        for j in range(m):
            classe = imgc[i,j]
            imgaplat[i,j,:] = lBary[classe]
    return imgaplat
```

⋆ Mise à jour des barycentres

On doit ensuite être capable de mettre à jour les coordonnées des barycentres à partir des classes.

Il s'agit donc d'écrire une fonction d'entête :

```
def calculebary(img, imgc, K):
    """
    entrée: img = array 3-d
                = image couleur originale
            imgc = array 2d de même taille que img
                 = imgc[i,j] est un int de [|0, K-1 |]
                   donnant la classe du pixel (i,j)
            K = int = nbr de classe
    sortie: lBary = list de triplet de int
                  = coordonnées des K barycentres
    """
```

Pour cela, on utilise la technique usuelle de calcul de somme, mais pour une liste de triplets. On initialise donc une liste de triplets de longueur K à 0, ainsi qu'une liste d'effectif (toujours de longueur K) pour chaque classe à 0.

On balaie ensuite tous les pixels. Pour chaque pixel, on regarde sa classe (lue dans `imgc`) et on ajoute sa couleur (lue dans `img`) à la somme de la classe correspondante et on incrémente l'effectif de la classe correspondante.

Ensuite, on n'a plus qu'à diviser la somme par l'effectif pour faire la moyenne.

REM : contrairement à l'algorithme du cours, on peut avoir des effectifs nuls à la fin, c'est-à-dire qu'un groupe n'existe plus puisque l'on a choisi d'initialiser les groupes sur des valeurs qui ne sont pas atteintes. Dans ce cas, on mettra au choix :

- soit sa valeur à `[1000, 1000, 1000]`. Dans ce cas, on s'assure qu'il n'est jamais utilisé,

- ou on lui donne une valeur aléatoire, de manière à relancer l'algorithme.

✯ Test de la convergence

L'algorithme s'arrête lorsque les groupes n'évoluent plus.

On propose de comparer les coordonnées des barycentres et d'utiliser la fonction :

```
def distanceList(L1, L2):
    """
    entrée: L1 et L2 deux listes de triplets  de même longueur
    sortie: distance entre les listes
    """

    D = [distCouleur(L1[i], L2[i]) for i in range(len(L1))]
    return sum(D)
```

qui permet de mesurer la distance entre deux listes de barycentres.

Si les barycentres sont suffisamment proches, on considère que l'algorithme a convergé. On pourra considérer que si la distance entre deux listes de barycentres est inférieure à 3 l'algorithme a convergé ou attendre jusqu'à ce que cette distance soit nulle.

En utilisant cette fonction et les précédentes, écrire une fonction qui permet de réaliser l'algorithme des K-means jusqu'à convergence. L'entête est :

```
def aplati(img, K ):
    """
    entrée: img = array 3-d
                = image couleur originale
            K = int = nbr de classe
    sortie: imgc = array 2d de même taille que img
                 = imgc[i,j] est un int de [|0, K-1 |]
                   donnant la classe du pixel (i,j)
             lBary = list de triplet de int
                   = coordonnées des K barycentres

    """
```

Tester avec le script et les images fournies.

4 — Programmation dynamique

La programmation dynamique est une technique permettant de résoudre certaines classes de problèmes de calcul de minimisation.

On va la présenter sur quelques exemples, en essayant de dégager une structure générale.

I Exemple modèle

I.1 Description du problème

On imagine un système qui peut être dans p états possibles. Cet état sera noté avec un entier de $[\![0, p-1]\!]$.

On étudie ce système au temps $t_0, \ldots, t_N$. Entre deux temps consécutifs, le système peut changer d'état. On a donc N transitions.

Le coût de changement d'un état dépend du temps et de l'état. On note $f(k,x,y)$ le coût pour passer de l'état x à l'état y entre le temps t_k et le temps t_{k+1}. Il est possible que $f(k,x,x) > 0$, autrement dit que rester dans le même état x ait un coût.

Les valeurs $(f(k,x,y))$ sont connues.

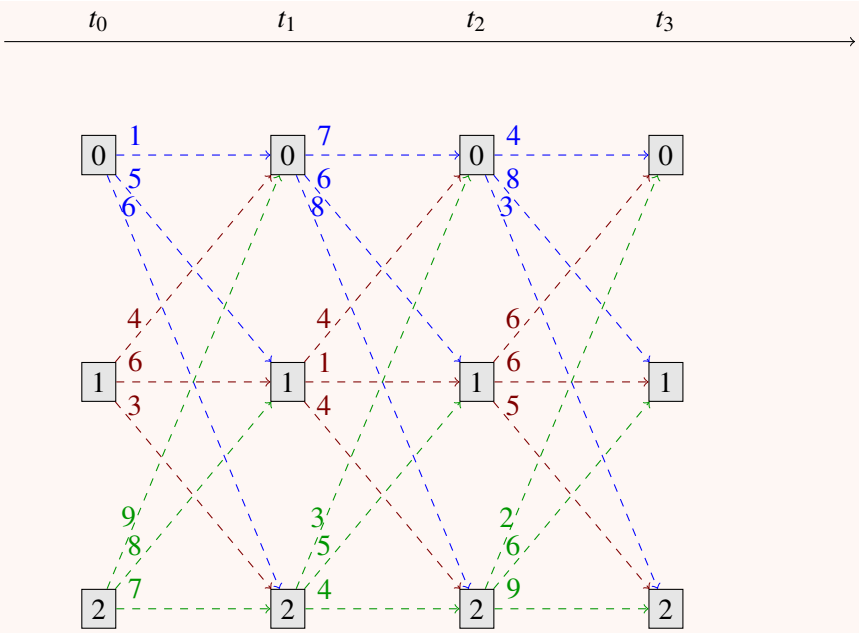
Le problème de minimisation est le suivant :

Trouver une suite d'états pour le système $x_0, x_1, \ldots x_N$ qui minimise le coût total :

$$C(x_0, \ldots, x_N) = \sum_{i=0}^{N-1} f(i, x_i, x_{i+1})$$

Cela revient à un graphe ou à chacun des temps, on a p états, chaque états est reliés à tous les états au temps suivants, y compris lui-même. La valeurs de l'arrête reliant l'état x et l'état y à la transition k est $f(k,x,y)$.

> Voici un exemple de problème avec 3 états et 3 temps : le chiffre indiqué sur une flèche indique le coût de la transition (tiré d'un livre)



On voit deux chemins minimaux de coût 9 : `(1,2,2,0)` et `(1,2,0,2)`.

Bien comprendre que l'on dispose à chaque transitions de p^2 flèches de transitions, au final on a donc Np^2 données.

On cherche donc un chemin de coût minimal dans ce graphe. c'est-à-dire :

$$\min_{x_0,\dots,x_n} c(x_0,\dots,x_n)$$

Le calcul du minimum se fait sur tous les chemins $((x_0,\dots,x_n))$, il y en a un nombre fini (c'est le nombre de liste de longueur $N+1$ de $[\![0,p-1]\!]$, c'est donc bien un minimum.

I.2 Équation de Bellman

Pour résoudre ce problème, un première idée est de parcourir tous les chemins pour trouver celui de coût minimal. Or il y a p^{N+1} chemins possibles, ce qui est trop long pour du temps de calcul raisonnable.

Une propriété des chemins de coût minimal permet en fait un calcul plus facile.

Considérons pour $x \in [\![0,p-1]\!]$:

$$\begin{aligned} c_k(x) &= \min_{(x_0,\dots,x_{k-1},x_k)\in[\![0,p-1]\!]^{k+1} \text{ avec } x_k=x} \left(\sum_{i=0}^{k-1} f(i,x_i,x_{i+1})\right) \\ &= \min_{x_0,\dots,x_{k-1}\in[\![0,p-1]\!]^k} \left(\sum_{i=0}^{k-2} f(i,x_i,x_{i+1}) + f(k-1,x_{k-1},x)\right) \end{aligned}$$

C'est le coût du chemin minimal pour arriver au temps k à l'état x.

En effet, on prend le minimum sur tous les chemins $(x_0,\dots,x_{k-1},x)$ de longueur $k+1$ qui se termine en x au temps k.

On parle de **sous-problème optimal**. La programmation dynamique consiste à exploiter cette structure de sous-problème optimal.

On peut calculer « à la main » les valeurs de $c_k(x)$ sur l'exemple :

	t_1	t_2	t_3
état 0	$c_1(0) = 1$ (0)	$c_2(0) = 6$ (2)	$c_3(0) = 9$ (2)
état 1	$c_1(1) = 5$ (0)	$c_2(1) = 6$ (1)	$c_3(1) = 12$ (1)
état 2	$c_1(2) = 3$ (1)	$c_2(2) = 7$ (2)	$c_3(2) = 9$(0)

On a mis entre parenthèse le y qui réalise le minimum.

On a alors :

- Le coût cherché est $\min_{x\in[\![0,p-1]\!]} c_N(x)$. C'est moins coûteux des chemins minimaux de longueur N. Autrement dit, on regarde pour tous les états d'arrivée, lequel est atteint avec un chemin minimal le moins coûteux.
- $c_0(x) = 0$ pour tout x (c'est le coût pour partir du point x).
- On a une relation dite **équation de Bellman** :

$$\forall k \in [\![1,N]\!],\ \forall x \in [\![0,p-1]\!],\ c_k(x) = \min_{y\in[\![0,p-1]\!]} (c_{k-1}(y) + f(k-1,y,x))$$

Montrons cette dernière égalité pour la comprendre.

Pour cela, on considère un chemin de coût minimal de longueur k qui termine à l'état x, on le note : $(x_0,\dots,x_{k-2},y,x)$ en notant donc $y \in [\![0,p-1]\!]$, l'état au temps $k-1$. Pour ce chemin, on a :

$$f(0,x_0,x_1) + \cdots + f(k-3,x_{k-3},x_{k-2}) + f(k-2,x_{k-2},y) + f(k-1,y,x) = c_k(x)$$

puisque c'est un chemin de coût minimal. D'autre chemins peuvent avoir le même coût mais aucun ne peut avoir un coût inférieur.

On va montrer que le sous-chemin extrait : $(x_0,\dots,x_{k-2},y)$ de longueur $k-1$ pour arriver à y est en fait de coût minimal parmi tous les chemins qui finissent à y au temps $k-1$. Pour cela on va raisonner par l'absurde, en supposant qu'il existe une amélioration de ce chemin, on note $(\tilde{x_0},\dots,\tilde{x}_{k-2},y)$ un autre chemin (toujours de longueur $k-1$ pour arriver à y) de coût strictement inférieur. On a alors :

$$\begin{aligned} &f(0,\tilde{x}_0,\tilde{x}_1) + \cdots + f(k-3,\tilde{x}_{k-3},\tilde{x}_{k-2}) + f(k-2,\tilde{x}_{k-2},y) < \\ &\quad f(0,x_0,x_1) + \cdots + f(k-3,x_{k-3},x_{k-2}) + f(k-2,x_{k-2},y) \end{aligned}$$

Alors, le chemin $(\tilde{x_0},\dots,\tilde{x}_{k-2},y,x)$ de longueur k permet d'arriver en x et ce chemin a un coût strictement inférieur à celui de $(x_0,\dots,x_{k-2},y,x)$, puisque :

$$\begin{aligned} &f(0,\tilde{x}_0,\tilde{x}_1) + \cdots + f(k-3,\tilde{x}_{k-3},\tilde{x}_{k-2}) + f(k-2,\tilde{x}_{k-2},y) + f(k-1,y,x) < \\ &\quad f(0,x_0,x_1) + \cdots + f(k-3,x_{k-3},x_{k-2}) + f(k-2,x_{k-2},y) + f(k-1,y,x) = c_k(x) \end{aligned}$$

ce qui n'est pas possible puisque le chemin $(x_0,\dots,x_{k-2},y,x)$ est optimal.

Ainsi, $(x_0,\dots,x_{k-2},y)$ est un chemin optimal pour arriver à y au temps $k-1$. Et donc :

$$f(0,x_0,x_1) + \cdots + f(k-2,x_{k-3},x_{k-2}) + f(k-1,x_{k-2},y) = c_{k-1}(y)$$

Ainsi, on peut écrire :

$$c_k(x) = c_{k-1}(y) + f(k-1,y,x)$$

Supposons maintenant par l'absurde qu'il existe un un $\tilde{y}$, tel que

$$c_{k-1}(\tilde{y}) + f(k-1,\tilde{y},x) < c_{k-1}(y) + f(k-1,y,x)$$

Considérons alors un chemin minimal permettant d'arriver à $\tilde{y}$ au temps $k-1$. On le note : $(\tilde{x_0},\ldots,\tilde{x}_{k-2},\tilde{y})$. On ajoute x à la fin ce qui donne : $(\tilde{x_0},\ldots,\tilde{x}_{k-2},\tilde{y},x)$, chemin de longueur k permettant d'arriver à x. Ce chemin aurait alors un coût :

$$c_{k-1}(\tilde{y}) + f(k-1,\tilde{y},x) < c_{k-1}(y) + f(k-1,y,x) = c_k(x)$$

Le chemin $(x_0,\ldots,x_{k-2},\tilde{y},x)$ aurait alors un coût inférieur strictement à celui de $(x_0,\ldots,x_{k-2},y,x)$ ce qui n'est pas possible. Donc y minimise l'expression : $c_{k-1}(y) + f(k-1,y,x)$ On peut dire que y est le meilleur des états au temps $k-1$ pour arriver à x au temps k, le meilleur prédécesseur.

Sur l'exemple à partir du tableau précédent, on voit que l'on peut retrouver le coût minimal 9 en prenant le minimum de la dernière colonne. Ensuite, on peut reconstruire le chemin optimal en remontant en arrière et en prenant toujours comme prédécesseur, celui qui réalise le minimum . En partant de $c_3(0) = 9$, on reconstruit un chemin qui finit en 0 au temps 3 : c'est 1,2,2,0. En partant de $c_3(2) = 9$, on reconstruit un chemin qui finit en 2 au temps 3 : c'est 1,2,0,2.

I.3 Calcul de chemin minimal de bas en haut

La première méthode pour calculer le chemin minimal est donc de calculer les $(c_k(x))_{k\in[\![0,N]\!],x\in[\![0,p-1]\!]}$ avec une boucle for.

Puisque l'on connaît les valeurs pour $k = 0$ ainsi que la relation de Bellman qui donne la relation de récurrence.

L'algorithme consiste donc à calculer la valeur de $c_k(x)$ pour chaque temps k et chaque état x, ainsi qu'à conserver en mémoire le meilleur prédécesseur à x au temps k

Pour cela :

- on initialise les valeurs à 0 pour $k = 0$.
- pour chaque temps k et chaque état x on cherche l'état y qui minimise $c_{k-1}(y) + f(k-1,y,x)$.
- La valeur de ce minimum donne la valeur de $c_k(x)$, l'indice de ce minimum (ie le y qui réalise ce minimum) est le meilleur prédécesseur à x au temps k.
- Une fois que tout cela est fait, on calcule le meilleur état d'arrivée, ie le x qui minimise $c_N(x)$. Cet état est la dernière étape de notre chemin, ie x_N.
- On considère le prédécesseur de x_N qui est donc x_{N-1}, puis on remonte ainsi, jusqu'à x_0. On peut ainsi reconstruire le chemin $(x_0,\ldots,x_N)$ de coût minimal.

Une manière de coder est de créer un conteneur C qui contient une liste de longueur $N+1$, de listes de longueur p, de couple `(cout, etat)`. Ainsi, `C[k][x]` est donc un couple `(cout, etat)`, avec `cout` un float et `etat` un int. C'est le coût pour arriver en x au temps k ainsi que le prédécesseur de x au temps $k-1$.

L'instruction :

```
[ C[k-1][y][0] + f(k-1, y, x) for y in range(p)]
```

permet de construire la liste de réels $(c_{k-1}(y)+f(k-1,y,x))$ pour tous les états y. C'est sur cette liste que l'on calcule le minimum (valeur de $c_k(x)$) et l'indice du minimum (le meilleur prédécesseur).

On code à part une fonction `minEtindmin` qui prend en entrée une liste de `float` et qui revoir un couple `(valeur, indice)` valeur et indice du minimum de cette liste.

Le chemin est stockée sous la forme d'une liste $(x_0,\dots,x_N)$. On construit cette liste en partant de la fin.

Un exemple de code avec ce choix est :

- Pour construire la liste C :

```
C = [ [ [0, None] for i in range(p)] for j  in range(N+1)]
for k in range(1,N+1):
  for x in range(p):
    C[k][x] = minEtindmin(
            [ C[k-1][y][0] + f(k-1, y, x) for y in range(p)] )
```

- Pour calculer le coût minimal et le point d'arrivée :

```
cout, xN = minEtindmin([ C[N][x][0] for x in range(p)] )
```

- Pour reconstruire le chemine :

```
chemin = [None]*(N+1)
chemin[N] = xN
for i in range(N-1, -1, -1):
  chemin[i] = C[i+1][chemin[i+1]][1]
```

La complexité de l'algorithme se calcule facilement

- Pour chaque k dans $[\![1,N+1]\!]$,
- Pour chaque x dans $[\![0,p-1]\!]$,
- on calcule la liste :

```
[ C[k-1][y][0] + f(k-1, y, x) for y in range(p)]
```

  ce qui fait p appel à f et p additions. Donc $2p$ opérations.
- on calcule le minimum, et l'indice du minimum de cette liste de longueur p, ce qui fait p comparaisons (rappel : une comparaison est beaucoup plus rapide qu'une addition).
- il reste un dernier minimum à calculer soit p opérations.

Cela fait donc $\Theta(p^2\times N)$ opérations.

R Avec la méthode de programmation que l'on a mis en place ici, le coût en mémoire est élevé : on stocke un tableau de $p\times N$ couples. On peut éventuellement, diminuer ce coût en programmant différemment et en écrasant à chaque itération les valeurs de $c_k(x)$.

I.4 Calcul récursif du chemin minimal

On peut aussi procéder de manière récursive : on n'écrit plus de boucle `for` sur k, mais on laisse les appels récursifs successifs fonctionner. Pour éviter les trop nombreux appels récursifs, on fait appel au dictionnaire : on stocke les valeurs de $C[k][x]$ dans un dictionnaire.

Si cette valeur n'existe pas, on la calcule par un appel récursif, si cette valeur existe, on revoie cette valeur.

- On utilise donc un dictionnaire indexé sur les tuples (k,x).
- On initialise ce dictionnaire en donnant la valeur `[0, None]` aux couples $(0,x)$ pour $x \in [\![0, p-1]\!]$.
- On crée une fonction récursive `calculeC` qui calcule `C[(k,x)]` ainsi :
  - Si l'index (k,x) est déjà présent dans le dictionnaire C, on renvoie sa valeur,
  - Sinon on la calcule avec l'équation de Bellman :

$$c_k(x) = \min_{y \in [\![0,p-1]\!]} (c_{k-1}(y) + f(k-1,y,x))$$

    Cela impose donc de calculer $c_{k-1}(y)$, ie `calculeC[(k-1,x)]`, c'est donc un appel récursif à la fonction. On récupère non seulement la valeur `cout` mais aussi l'indice `etat` du minimum, ie le meilleur prédécesseur. Le couple `(cout, etat)` est stockée dans le dictionnaire C qui est modifié et retournée par la fonction
- On appelle cette fonction récursive pour calculer `C[(n,x)]`.
- Il reste ensuite à déterminer comme précédement le meilleur état d'arrivée, ie le x qui minimise $c_N(x)$. Cet état est la dernière étape de notre chemin, ie x_N. Puis de reconstruire le chemin $(x_0, \dots, x_N)$ de coût minimal.

Avec ce choix un exemple de code peut être : Pour l'initialisation de C :

```
C = {}
for x in range(p):
  C[(0,x)] = [0, None]
```

Pour le calcul récursif, on crée une fonction dans la fonction. Cette fonction peut avoir accès à l'espace appelant et donc au dictionnaire C.

```
def calculeC(k,x):
  """
  fonction récursive qui remplit le dictionnaire
  """
  if (k,x) not in C:
    C[(k,x)] = minEtindmin( [ calculeC(k-1,y)[0] + f(k-1, y, x) for y in range(p)] )
  return C[(k,x)]

# l'appel à la fonction récursive se fait ici:
for y in range(p):
  calculeC(p,y)
```

Enfin, il reste à calculer le cout minimal et reconstruire le chemin

```
    # une fois C rempli, on calcule le cout minimal
    cout, xN = minEtindmin([ C[(p,y)][0] for y in range(p)] )

    # puis on renstruit le chemin complet
    chemin = [None]*(N+1)
```

```
    chemin[N] = xN
    for i in range(N-1, -1, -1):
        chemin[i] = C[( i+1, chemin[i+1]) ][1]
```

II Cas général

À partir de l'exemple précédent, on peut dégager une structure de problèmes pouvant être résolus par programmation dynamique :

- Il s'agit de problèmes de minimisation : le problème consiste à chercher un objet de coût minimal parmi une grande quantité d'objets.
  On a un ensemble E fini mais très grand et on cherche la valeur de $\min_{x\in E} f(x)$ et le x qui réalise le minimum. La fonction f est connue.
  Dans notre exemple : le chemin de coût minimal parmi tous les chemins possibles.
- Les solutions du problème de départ possèdent une *propriété de sous-structure optimale.*
  Cela signifie que le x qui réalise le minimum de $\min_{x\in E} f(x)$ contient aussi une solution y de sous-problèmes plus simple (de taille plus petits).
  Ainsi, on peut découper le problème de départ en sous-problèmes plus simple (plus petits). Jusqu'à un problème minimal qui a une solution triviale (initialisation).
  Dans notre exemple : un chemin de coût minimal est minimal à toutes les étapes.
  Le problème initial est le plus gros des sous-problèmes (le dernier). Tandis que le plus petit des sous-problème admet une solution triviale.
  Dans notre exemple, on connaît $c_0(x)$ pour tout x (qui est nul).
- Ces sous-problèmes ne sont pas indépendants, mais au contraire entremêlés. Il y a un *chevauchement des sous-problème.*
  Si l'on veut résoudre un sous-problème d'une taille donné, on peut le faire facilement en utilisant les solutions des sous problèmes d'une taille plus petite.
  Ou dans l'autre sens : en partant des problèmes de taille petites, on peut résoudre des problèmes de taille plus grande.
  On a donc une *équation de Bellman* qui est donc une formule de récurrence permettant de résoudre les sous-problème un par un dans un certain ordre.
  Dans notre exemple : si on connaît les valeurs $c_{k-1}(y)$ pour $y \in [\![0, p-1]\!]$, ie si on a résolu le $k-1$ième sous-problème, alors il est facile de calculer les valeur de $(c_k(x))$ avec l'équation de Bellman.

On peut utiliser la programmation dynamique de deux manières :

- Soit par programmation itérative (approche bas en haut) : on part du sous-problème le plus petit et on utilise une boucle `for` pour résoudre des problèmes de plus en plus gros.
- soit par programmation récursive : on utilise des appels récursifs. On part alors du problème le plus gros et on le découpe en sous-problème plus petit, jusqu'au problème trivial.
  Dans ce cas, pour éviter les trop nombreux appels récursifs, on utilise la mémoï-

sation : on stocke les valeurs calculées dans un dictionnaire.

Il faut favoriser la récursivité lorsqu'il n'est pas nécessaire de résoudre tous les sous-problème.

Par exemple, quand le problème initial est de taille n et nécessite de résoudre les problèmes pour tous les diviseurs de n.

III Deuxième exemple : algorithme de Roy-Floyd-Warshall

On considère un graphe valué dont les sommets sont les entiers $[\![0,n-1]\!]$. Il est représenté par sa **matrice d'adjacence** W.

On dispose donc d'une matrice W, avec :

$$W_{i,j} = \begin{cases} +\infty & \text{si } i \text{ n'est pas relié à } j \\ w_{i,j} & \text{longueur de l'arc de } i \text{ vers } j \end{cases}$$

Soit i et j deux sommets. Un **chemin** qui relie i à j est une suite finie de sommets reliés qui commence à i et finit à j. Autrement dit, c'est une liste

$$p = (i_0, \dots, i_l) \text{ avec } i_0 = i \text{ et } i_l = j$$

avec $\forall s \in [\![0,l-1]\!], w_{i_s,i_{s+1}} \neq +\infty$ (autrement dit, chaque sommet est bien relié au suivant).

On note de plus $C_{i,j}$ l'ensemble des chemins de i à j.

La **longueur du chemin** $p = (i_0, \dots, i_l)$ est :

$$L(p) = \sum_{s=0}^{l-1} w_{i_s,i_{s+1}}$$

C'est la somme des longueurs des arrêtes.

On s'intéresse à la recherche du chemin le plus court entre deux sommets i et j, ie la longueur la plus faible.

On cherche donc :

$$\mathscr{L}(i,j) = \min_{p \in C_{i,j}} L(p)$$

On considère un chemin optimal reliant i à j, que l'on note :

$$p = (i_0, \dots, i_{s-1}, i_s, i_{s+1}, \dots, i_l)$$

On extrait le sous-chemin $(i_0, \dots, i_{s-1}, i_s)$, alors ce chemin est optimal pour relier i_0 (ie i) à i_s, car si on pouvait améliorer cette partie du trajet, on pourrait obtenir un meilleur chemin optimal pour relier i à j. De même, le sous-chemin $(i_s, i_{s+1}, \dots, i_l)$ est optimal pour relier i_s à i_l (ie j).

À partir de cette idée, on considère les ensembles $\left(C_{i,j}^k\right)_{(i,j,k)\in[\![0,n-1]\!]^3}$ qui sont définis comme l'ensemble (éventuellement vide) des chemins qui relient les sommets i et j et dont tous les sommets intermédiaires sont dans $[\![0,k-1]\!]$.

Ainsi, un chemin $p = (i_0, \dots, i_l)$ est dans $C^k_{i,j}$ lorsque :

$$i = i_0,\ j = i_l,\ \text{ et } \forall s \in [\![1, l-1]\!],\ i_s \in [\![0, k-1]\!]$$

Avec comme convention que : $\left(C^0_{i,j}\right)$ est l'ensemble (éventuellement vide) des arrêtes qui relient i à j.

On peut remarquer de plus que $\left(C^n_{i,j}\right)$ représente les chemins (quelconques) qui relient i à j. Ainsi, le problème posé revient à chercher le chemin de longueur minimal dans $\left(C^n_{i,j}\right)$.

On note donc :

$$\mathscr{L}^k(i,j) = \min_{p \in C^k_{i,j}} L(p)$$

Ainsi, $\mathscr{L}^k(i,j)$ est la longueur minimale d'un chemin qui relie i à j en ne passant que par des sommets de $[\![0, k-1]\!]$. Par convention, ce min est égal à $+\infty$ si $C^k_{i,j}$ est vide.

Le problème initial est de trouver $\mathscr{L}^n(i,j)$.

Considérons alors $p = (i_0, \dots, i_l)$ un chemin optimal permettant de relier i à j en ne passant que par des sommets de $[\![0, k-1]\!]$. Ainsi, $p \in C^k_{i,j}$ et la longueur de p est égale à $\mathscr{L}^k(i,j)$.

On a alors deux possibilités :

- si $k-1$ est l'un des sommets intermédiaires, alors on peut écrire :

  $$p = (i_0, \dots, i_{s-1}, i_s, i_{s+1}, \dots, i_l) \text{ avec } i_0 = i,\ i_s = k-1,\ \text{et } i_l = j$$

  Le chemin p est ainsi la concaténation de p_1 et p_2, avec :

  $$p_1 = (i_0, \dots, i_{s-1}, i_s)$$
  $$\text{et } p_2 = (i_s, i_{s+1}, \dots, i_l)$$

  On voit que p_1 est un chemin qui relie i à $k-1$ et p_2 un chemin qui relie $k-1$ à j. Comme on l'a vu, p_1 et p_2 sont optimaux et de plus ont leurs sommets intermédiaires dans $[\![0, k-2]\!]$ (ces chemins ne passent pas en cours de route par $k-1$).
  Par définition de la longueur : $L(p) = L(p_1) + L(p_2)$ comme il s'agit de chemin optimaux, cela s'écrit :

  $$\mathscr{L}^k(i,j) = \mathscr{L}^{k-1}(i, k-1) + \mathscr{L}^{k-1}(k-1, j)$$

- Sinon, c'est que p ne passe pas par $k-1$ et donc que tous ses sommets intermédiaires sont dans $[\![0, k-2]\!]$, ie p est en fait dans $C^{k-1}_{i,j}$. Dans ce cas :

  $$\mathscr{L}^k(i,j) = \mathscr{L}^{k-1}(i,j)$$

Comme l'une et une seules de ces possibilités se produit, cela donne la relation :

$$\mathscr{L}^k(i,j) = \min\left(\mathscr{L}^{k-1}(i,k-1) + \mathscr{L}^{k-1}(k-1,j), \mathscr{L}^{k-1}(i,j)\right)$$

On a donc une relation de Bellman. Pour l'initialisation, on a :

$$\mathscr{L}^0(i,j) = \begin{cases} +\infty & \text{si } i \text{ et } j \text{ ne sont pas reliés} \\ w_{i,j} & \text{sinon} \end{cases}$$

De plus, le travail ci-dessus permet de calculer la longueur minimale $\mathscr{L}^k(i,j)$ mais aussi le chemin minimal. Pour i (point de départ), j (point d'arrivée) et k (plus grande valeur de sommet intermédiaire) fixés, on a deux choix :

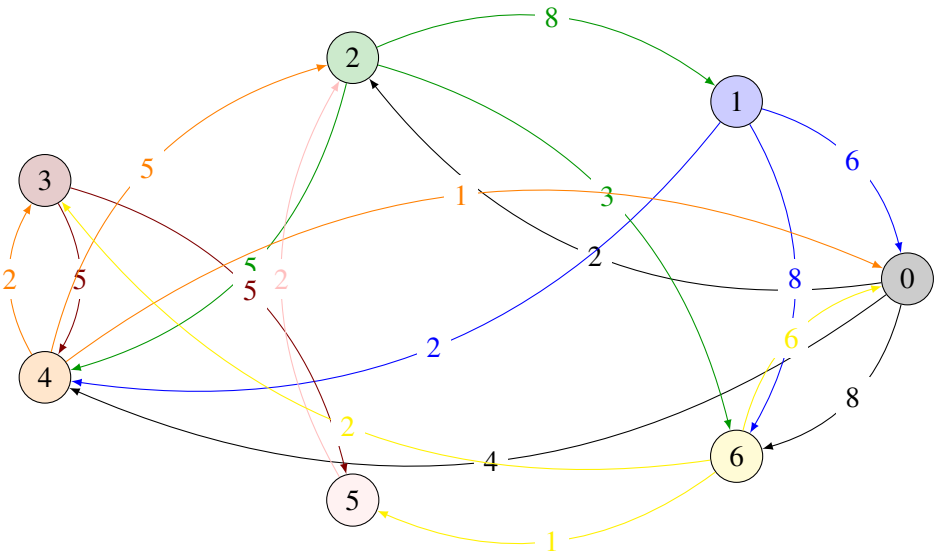
- Soit $\mathscr{L}^{k-1}(i,j) \leqslant \mathscr{L}^{k-1}(i,k-1) + \mathscr{L}^{k-1}(k-1,j)$, dans ce cas, le chemin optimal de $C^k_{i,j}$ ne passe pas par $k-1$, c'est déjà le chemin optimal qui permet de réaliser $\mathscr{L}^{k-1}(i,j)$.
- Sinon, c'est que le chemin optimal de $C^k_{i,j}$ passe par $k-1$, et ce chemin est alors la concaténation de p_1 et p_2, où p_1 est le chemin optimal de $C^{k-1}_{i,k-1}$ et p_2 le chemin optimal de $C^{k-1}_{k-1,j}$.

⋆ Exemple de donnée

Voici par exemple, une matrice d'adjacence et le graphe associé :

```
W = array(
    [[INF, INF, 2, INF, 4 , INF, 8],
     [6, INF, INF, INF, 2 , INF, 8],
     [INF, 8, INF, INF, 5 , INF, 3],
     [INF, INF, INF, INF, 5 , 5, INF],
     [1, INF, 5, 2, INF , INF, INF],
     [INF, INF, 2, INF, INF, INF , INF],
     [6, INF, INF, 2, INF, 1 , INF]
     ])
```

Le graphe associé est :



Les distances optimales sont :

	0	1	2	3	4	5	6
0	5	10	2	6	4	6	5
1	3	13	5	4	2	9	8
2	6	8	6	5	5	4	3
3	6	15	7	7	5	5	10
4	1	11	3	2	5	7	6
5	8	10	2	7	7	6	5
6	6	11	3	2	7	1	6

Les chemins optimaux sont :

	0	1	2	3	4	5
0	`[0, 4, 0]`	`[0, 2, 1]`	`[0, 2]`	`[0, 4, 3]`	`[0, 4]`	`[0, 2, 6, 5]`
1	`[1, 4, 0]`	`[1, 4, 0, 2, 1]`	`[1, 4, 0, 2]`	`[1, 4, 3]`	`[1, 4]`	`[1, 4, 0, 2, 6, 5]`
2	`[2, 4, 0]`	`[2, 1]`	`[2, 6, 5, 2]`	`[2, 6, 3]`	`[2, 4]`	`[2, 6, 5]`
3	`[3, 4, 0]`	`[3, 5, 2, 1]`	`[3, 5, 2]`	`[3, 4, 3]`	`[3, 4]`	`[3, 5]`
4	`[4, 0]`	`[4, 0, 2, 1]`	`[4, 0, 2]`	`[4, 3]`	`[4, 0, 4]`	`[4, 0, 2, 6, 5]`
5	`[5, 2, 4, 0]`	`[5, 2, 1]`	`[5, 2]`	`[5, 2, 6, 3]`	`[5, 2, 4]`	`[5, 2, 6, 5]`
6	`[6, 0]`	`[6, 5, 2, 1]`	`[6, 5, 2]`	`[6, 3]`	`[6, 3, 4]`	`[6, 5]`

LA SUITE EN TP !

Annexe programmes complets programmation dynamique

Pelletier Sylvain PSI, LMSC

⋆ Recherche de chemin de coût minimal

```python
def minEtindmin(L):
    """
    entrée: L = list de float
    sortie: m = float = minimum de la liste L
            i = int = indice du minimum
    """
    m = min(L)
    return m, L.index(m)


def f(k, x, y):
    """
    entrée: k = int  de [|0, N-1|]
              = numéro de l'étape
            (x,y) = deux int de [|0, p-1|]
                  = deux états
    sortie: float
            cout de transition de l'état x
            à l'état y entre les temps k et k+1

    """
    # ici on utilise une liste de liste de liste
    # pour stocker les coûts
    mat = [
        [[1,5,6], [4,6,3], [9,8,7]], # transition  0 -> 1
        [[7,6,8], [4,1,4], [3,5,4]], # transition  1 -> 2
        [[4,8,3], [6,6,5], [2,6,9]] # transition  2 -> 3
    ]

    return mat[k][x][y]

def cheminMinimal(N , p, f):
    """
    entrée: N = int
              = nbr de transition
              les temps vont de 0 à N
            p = int
              = nbr d'états
            f = fct
              = f(k,x,y) est le coût de la
                transition de x vers y au temps k
    return: chemin = list de int de lontueur N+1
              = (x0, x1 , ... , xN) chemin de cout minimal
            coutChemin = float
                       = cout de ce chemin
    """


    # C contient une liste  de longueur N+1 de liste de longueur p de couple (cout, etat)
    # C[k][x] est donc un couple (cout, etat)
    # avec cout un float, etat un int
    # c'est le cout pour arriver en x ainsi que le prédécesseur de x

```

```
    C = [ [ [0, None] for i in range(p)] for j  in range(N+1)]
    for k in range(1,N+1):
        for x in range(p):
            C[k][x] = minEtindmin([ C[k-1][y][0] + f(k-1, y, x) for y in range(p)] )

    for k in range(N, -1, -1):
        for x in range(p):
            print("à l'étape", k, "cout minimal pour arriver à",x,
                  ":", C[k][x][0], " prédécesseur:", C[k][x][1])



    # une fois C rempli, on calcule le cout minimal
    cout, xN = minEtindmin([ C[N][x][0] for x in range(p)] )

    # puis on renstruit le chemin complet
    chemin = [None]*(N+1)
    chemin[N] = xN
    for i in range(N-1, -1, -1):
        chemin[i] = C[i+1][chemin[i+1]][1]

    return chemin, cout


def cheminMinimalRecurssif(N , p, f):
    """
    Version récursive par mémoïsation

    entrée: N = int
              = nbr de transition
              les temps vont de 0 à N
            p = int
              = nbr d'états
            f = fct
              = f(k,x,y) est le coût de la
                transition de x vers y au temps k
    return: chemin = list de int de lontueur N+1
              = (x0, x1 , ... , xN) chemin de cout minimal
            coutChemin = float
                       = cout de ce chemin
    """

    # C est un dictionnaire indexé sur les tuples  (k,x).
    # la valeur de   C[(k,x)] est un couple (cout, etat)
    # avec cout un float, etat un int
    # c'est le cout pour arriver en x ainsi que le prédécesseur de x

    # initialisation de C:
    C = {}
    for x in range(p):
        C[(0,x)] = [0, None]

    # calcul récursif de C
    def calculeC(k,x):
        """
        fonction récursive qui remplit le dictionnaire
        """
        if (k,x) not in C:
            C[(k,x)] = minEtindmin( [ calculeC(k-1,y)[0] + f(k-1, y, x) for y in range(p)] )
        return C[(k,x)]

    # l'appel à la fonction récursive se fait ici:
    for y in range(p):
        calculeC(N,y)
```

```
119
    # pour vérif
121     for k in range(N, -1, -1):
            for x in range(p):
123             print("à l'étape", k, "cout minimal pour arriver à",x,
                      ":", C[(k,x)][0], " prédécesseur:", C[(k,x)][1])
125
    # une fois C rempli, on calcule le cout minimal
127     cout, xN = minEtindmin([ C[(p,y)][0] for y in range(p)] )

129     # puis on renstruit le chemin complet
        chemin = [None]*(N+1)
131     chemin[N] = xN
        for i in range(N-1, -1, -1):
133         chemin[i] = C[( i+1, chemin[i+1]) ][1]

135     return chemin, cout


137




139
print("--- début du programme de programation dynamique ----")
141
print("--- test de la version non récursive ---")
143
chemin, cout = cheminMinimal(3 , 3, f)
145 print("cout minimal:", cout)
print("chemin minimal: ", chemin)
147
print("--- test de la version récursive ---")
149 chemin, cout = cheminMinimalRecurssif(3 , 3, f)
print("cout minimal:", cout)
151 print("chemin minimal: ", chemin)
```

Algorithme de remplissage Flood Fill

Pelletier Sylvain PSI, LMSC

Le but de ce projet est d'écrire une application de coloriage pour les jeunes enfants comme il en existe pour les tablettes :
- On présente à l'enfant une image à colorier, c'est-à-dire une image noir et blanc où les contours sont en noir,
- l'enfant choisit une couleur et un point dans l'image (un pixel donc)
- la machine remplit alors la totalité de la zone contenant le point choisi avec la couleur choisie.

⋆ Représentation des images numériques

Une **image numérique couleur** corresponds à la donnée de trois tableaux. Chaque case du tableau (appelée **pixel**) correspondant à la valeur en un point des **trois composantes chromatiques** de l'image : le rouge, le vert et le bleu.

Plus précisément, en python l'image de taille $n \times m$ sera représenté sous la forme d'un `3d-array` `img` de taille $n \times m \times 3$. La valeur de `img[i,j,0]` est le niveau de rouge du pixel `(i,j)`, la valeur de `img[i,j,1]` le niveau de vert, et celle de `img[i,j,2]` le niveau de bleu. On peut ainsi avoir accès et modifier la couleur du pixel `(i,j)` pour $i \in [\![0, n-1]\!]$ et $j \in [\![0, m-1]\!]$.

R Souvent, `img` est en fait un `3d-array` de taille $n \times m \times 4$. La quatrième composante chromatique est **la transparence** que l'on n'utilisera pas ici.

Parfois, on préfère travailler sur les trois composantes chromatiques (ie trois `2d-array`) plutôt que directement sur le `3d-array`. Cela revient à écrire : `R,G,B = img[:,:,0], img[:,:,1], img[:,:,2]`. On utilise alors `R[i,j]` pour avoir accès à la composante rouge du pixel `[i,j]`. Tout changement sur les `array` `R`, `G` et `B` se retrouve sur le `array` `img`.

Selon la manière dont l'image est codée et selon comment elle est lue par le logiciel, cet `array` contiendra :
- des entiers de $[\![0, 255]\!]$ codés sur $8bits$, c'est le type `unit8`.
- des réels de $[0, 1[$.

Dans la suite, on prends la convention que les valeur sont codées sous la forme de `uint8`. Si ce n'est pas le cas, il est facile d'adapter.

! Il y a un piège lorsqu'on manipule des `uint8` : si on fait la somme de deux entiers codés en `uint8`, on peut obtenir un entier supérieur à 255 et donc un dépassement de capacité ! Il faut les convertir en entier (avec `int`).

Lorsqu'on manipule les images, il est important de savoir dans lequel de ces cas on s'est placé. Pour vérifier, le plus simple est d'afficher la valeur et le type du pixel central comme cela est fait sur le script fourni.

Les différentes couleurs sont obtenues en donnant des valeur aux composantes (R,G,B) respectivement, niveau de rouge, vert et bleu. On obtient ainsi les différentes couleurs :

Le blanc corresponds à $(R,G,B) = (255,255,255)$,
le noir à $(R,G,B) = (0,0,0)$,
le bleu à $(R,G,B) = (0,0,255)$,
le gris moyen à $(R,G,B) = (128,128,128)$,
le jaune à $(R,G,B) = (255,255,0)$
le violet à $(R,G,B) = (255,0,255)$ etc.

En particulier, la luminosité d'une pixel corresponds à la somme des trois composantes. Plus le pixel est clair plus cette somme est importante.

En mélangeant les différents niveaux, on a $256^3 = 16777216$ couleurs différentes, ce qui corresponds à la résolution des écrans standards.

■ **Exemple III.1** On peut par exemple utiliser le array `img` de la manière suivante :
- pour savoir si un pixel (x,y) est noir, on peut simplement tester si la somme des trois composantes est faible :
  `int(img[x,y,0]) + int(img[x,y,1])+ int(img[x,y,2]) < SOM_NOIR`.
  Par exemple `SOM_NOIR = 10*3` est une bonne valeur,
- pour savoir si un pixel (x,y) est blanc, on peut simplement tester :
  `int(img[x,y,0]) + int(img[x,y,1])+ int(img[x,y,2]) > SOM_BLANC`.
  Par exemple `SOM_BLANC = 250*3` est une bonne valeur.
- Pour donner la couleur (r,g,b) à un pixel, où (r,g,b) est un triplet d'entier de $[\![0, 255]\!]$. Il suffit d'écrire :
  `img[x,y,0], img[x,y,1], img[x,y,2] = r,g,b`.

■

⋆ Modélisation

Pour modéliser l'application, on utilise la notion de composante connexe discrète (dans un tableau donc).
On considère l'ensemble des pixels de l'image qui est identifié comme l'ensemble $\mathscr{P} = [\![0, n-1]\!] \times [\![0, m-1]\!]$.
À une itération de l'algorithme, on note :

- $\mathscr{N}$ l'ensemble des pixels noirs, et $\overline{\mathscr{N}}$ l'ensemble des pixels qui ne sont pas noirs (peuvent être blanc dans l'image originale ou de couleur dans la suite).
- $\mathscr{B}$ l'ensemble des pixels blancs, et $\overline{\mathscr{B}}$ l'ensemble des pixels qui ne sont pas blancs.

Deux pixels $x = (i, j)$, $y = (i', j')$ sont voisins dans $\mathscr{P}$ si $|i - i'| + |j - j'| \leqslant 1$, c'est-à-dire si ils se touchent par un bords. On note alors $x \mathscr{V} y$. C'est une relation d'équivalence.

R Par convention, on considère qu'un pixel est son propre voisin, c'est-à-dire que $\mathscr{V}$ est réflexive.
C'est ici la 4 connexité que l'on utilise puisqu'un pixel situé au centre a 4 voisins. On peut aussi utiliser la 8-connexité, avec $\max(|i - i'|, |y - y'|) \leqslant 1$, c'est-à-dire si ils se touchent par un bords ou par un coin dans l'image.

On définit alors la relation : « *être dans la même composante connexe de blanc* », par la définition suivante : un pixel blanc $x \in \mathscr{B}$ est dans la même composante connexe que le pixel blanc $y \in \mathscr{B}$ si et seulement si

$$\exists p \in \mathbb{N}, (x_0, \dots, x_p) \in \mathscr{P}^p, \qquad \text{tels que } \begin{cases} x_0 = x \text{ et } x_p = y, \\ \forall i \in [\![0, p-1]\!],\ x_i \mathscr{V} x_{i+1} \\ \forall i \in [\![0, p]\!], x_i \in \mathscr{B}. \end{cases}$$

Autrement dit, si il existe un chemin dans l'image constitué de pixels voisins et tous blancs qui part de x et va jusqu'à y.
On note alors $x \mathscr{R} y$ et c'est encore une relation d'équivalence.

Ainsi modélisé, le problème peut s'écrire de la manière suivante : pour un pixel x donné trouver tous les pixels y tels que $x \mathscr{R} y$. On cherche donc la classe d'équivalence de x pour la relation $\mathscr{R}$.
Ce problème peut ainsi se généraliser à d'autres relations et ne sert donc pas uniquement aux applications de coloriage.

L'algorithme pour résoudre ce problème s'appelle Flood Fill.
Les buts de ce projets sont :

- Écrire un algorithme utilisant une pile pour résoudre ce problème,
- Ré-écrire le même programme en utilisant la récursivité.
- Évaluer la complexité en temps et en espace des deux algorithmes.

⋆ Algorithme non récursif

La fonction `floodFillPile` prends en entrée :

- l'image `img` (un `3d-array` donc),
- la couleur de remplissage (une liste de trois entiers `r,g,b` de $[\![0, 255]\!]$),
- le pixel (x, y) (deux entiers) sur lequel on a cliqué.

On utilise une pile de pixel (ie de couple d'entiers de $[\![0, n-1]\!] \times [\![0, m-1]\!]$) de la manière suivante :

- On initialise la pile avec le pixel (x, y).
- À chaque itération, on dépile un pixel de la pile si il est blanc, il passe à la couleur demandé et on empile tous les voisins.
- On continue ainsi jusqu'à ce que la pile soit vide.

Compléter le script fourni en écrivant la fonction `floodFillPile`

R Il y a donc un voisins pour chaque direction (nord, sud, ouest, est), pensez à indiquer dans vos commentaires quels voisins vous traitez. **Bien sûr : attention aux bords !** Certains pixels ont 4 voisins, d'autres 3 ou 2.

⋆ L'algorithme récursif

On va reprendre l'algorithme précédent sans utiliser de pile mais avec la récursivité.

Le principe de l'algorithme récursif est le suivant : on utilise une fonction `floodFillRec` qui prends en entrée :

- l'image `img` (un `3d-array` donc),
- la couleur de remplissage (une liste de trois entiers `r,g,b` de $[\![0, 255]\!]$),

- le pixel (x,y) (deux entiers) sur lequel on a cliqué.

Cette fonction ne sort aucune valeur mais modifie l'objet image.
Étant donné ces valeurs :
- si le pixel (x,y) n'est pas blanc (noir ou déjà colorié), la fonction s'arrête sans rien faire.
- sinon le pixel (x,y) est mis à la couleur demandé, puis la fonction s'appelle elle-même pour colorier les voisins.

Compléter le script fourni en écrivant la fonction `floodFillRec`. Tester.

✯ Les fichiers fournis

Vous disposez d'un script à compléter et d'une image de test.

Plus précisément, vous devez compléter les fonctions `floodFillPile` et `floodFillRec`, puis décommenter et modifier les instruction situées en dessous qui colorie les zones.

Le résultat obtenu s'affiche et peut être sauvegardé.

Utilisez les options d'exécution (`[F6]`) suivantes : « *dans un interpréteur dédié* », option « *interagir avec l'interpréteur après exécution* ».

Attention à la taille maximale de la pile de récursion. En particulier, expliquer pourquoi l'algorithme ne peut pas fonctionner depuis le pixel central.

Il est possible de modifier la taille de la pile d'appels de Python, mais cela peut rendre votre processus Python instable, surtout si vous lancez votre script depuis un IDE comme Spyder.

✯ Évaluation de la complexité en temps et en espace

Pour évaluer la complexité, on applique l'algorithme à une image carrée $[\![0,n-1]\!] \times [\![0,n-1]\!]$ en considérant que tous les pixels sont blancs.

On évalue alors le nombre de test effectué en fonction de n.

1. Évaluer le nombre de test avec la méthode récursive et le nombre d'appel de la fonction. À quoi sert l'instruction : `sys.setrecursionlimit(valeur)` ? Quelle valeur choisir ? Comment la modifier pour adapter cet algorithme à une autre taille d'image.
2. Évaluer le nombre de test avec la méthode non récursive, évaluer la taille de la pile avec la méthode non récursive.
3. Proposer quelques idées pour améliorer ces algorithmes.

Indiquez vos réponses sous la forme de commentaire (en « *doc string* ») à la fin de votre script.

Recherche de plus court chemin dans un graphe avec l'algorithme de Roy-Floyd-Warshall

Pelletier Sylvain PSI, LMSC

On considère un graphe valué dont les sommets sont les entiers $[\![0,n-1]\!]$. Il est représenté par sa **matrice d'adjacence** W.

On dispose donc d'une matrice W, avec :

$$W_{i,j} = \begin{cases} +\infty & \text{si } i \text{ n'est pas relié à } j \\ w_{i,j} & \text{longueur de l'arc de } i \text{ vers } j \end{cases}$$

Soit i et j deux sommets. Un **chemin** qui relie i à j est une suite finie de sommets reliés qui commence à i et finit à j. Autrement dit, c'est une liste

$$p = (i_0, \dots, i_l) \text{ avec } i_0 = i \text{ et } i_l = j$$

avec $\forall s \in [\![0,l-1]\!], w_{i_s,i_{s+1}} \neq +\infty$ (autrement dit, chaque sommet est bien relié au suivant).

On note de plus $C_{i,j}$ l'ensemble des chemins de i à j.

La **longueur du chemin** $p = (i_0, \dots, i_l)$ est :

$$L(p) = \sum_{s=0}^{l-1} w_{i_s,i_{s+1}}$$

C'est la somme des longueurs des arrêtes.

On s'intéresse à la recherche du chemin le plus court entre deux sommets i et j, ie la longueur la plus faible.

On cherche donc :

$$\mathscr{L}(i,j) = \min_{p \in C_{i,j}} L(p)$$

On considère un chemin optimal reliant i à j, que l'on note :

$$p = (i_0, \dots, i_{s-1}, i_s, i_{s+1}, \dots, i_l)$$

On extrait le sous-chemin $(i_0, \dots, i_{s-1}, i_s)$, alors ce chemin est optimal pour relier i_0 (ie i) à i_s, car si on pouvait améliorer cette partie du trajet, on pourrait obtenir un meilleur chemin optimal pour relier i à j. De même, le sous-chemin $(i_s, i_{s+1}, \dots, i_l)$ est optimal pour relier i_s à i_l (ie j).

À partir de cette idée, on considère les ensembles $\left(C_{i,j}^k\right)_{(i,j,k)\in[\![0,n-1]\!]^3}$ qui sont définis comme l'ensemble (éventuellement vide) des chemins qui relient les sommets i et j et dont tous les sommets intermédiaires sont dans $[\![0,k-1]\!]$.

Ainsi, un chemin $p = (i_0, \dots, i_l)$ est dans $C_{i,j}^k$ lorsque :

$$i = i_0,\ j = i_l,\ \text{ et } \forall s \in [\![1,l-1]\!],\ i_s \in [\![0,k-1]\!]$$

Avec comme convention que : $\left(C_{i,j}^0\right)$ est l'ensemble (éventuellement vide) des arrêtes qui relient i à j.

On peut remarquer de plus que $\left(C_{i,j}^n\right)$ représente les chemins (quelconques) qui relient i à j. Ainsi, le problème posé revient à chercher le chemin de longueur minimal dans $\left(C_{i,j}^n\right)$.

On note donc :

$$\mathscr{L}^k(i,j) = \min_{p \in C_{i,j}^k} L(p)$$

Ainsi, $\mathscr{L}^k(i,j)$ est la longueur minimale d'un chemin qui relie i à j en ne passant que par des sommets de $[\![0,k-1]\!]$. Par convention, ce min est égal à $+\infty$ si $C_{i,j}^k$ est vide.

Le problème initial est de trouver $\mathscr{L}^n(i,j)$.

Considérons alors $p = (i_0, \dots, i_l)$ un chemin optimal permettant de relier i à j en ne passant que par des sommets de $[\![0, k-1]\!]$. Ainsi, $p \in C^k_{i,j}$ et la longueur de p est égale à $\mathscr{L}^k(i,j)$.

On a alors deux possibilités :

- si $k-1$ est l'un des sommets intermédiaires, alors on peut écrire :

$$p = (i_0, \dots, i_{s-1}, i_s, i_{s+1}, \dots, i_l) \text{ avec } i_0 = i,\ i_s = k-1,\ \text{et } i_l = j$$

Le chemin p est ainsi la concaténation de p_1 et p_2, avec :

$$\begin{aligned} p_1 &= (i_0, \dots, i_{s-1}, i_s) \\ \text{et } p_2 &= (i_s, i_{s+1}, \dots, i_l) \end{aligned}$$

On voit que p_1 est un chemin qui relie i à $k-1$ et p_2 un chemin qui relie $k-1$ à j. Comme on l'a vu, p_1 et p_2 sont optimaux et de plus ont leurs sommets intermédiaires dans $[\![0, k-2]\!]$ (ces chemins ne passent pas en cours de route par $k-1$).
Par définition de la longueur : $L(p) = L(p_1) + L(p_2)$ comme il s'agit de chemin optimaux, cela s'écrit :

$$\mathscr{L}^k(i,j) = \mathscr{L}^{k-1}(i, k-1) + \mathscr{L}^{k-1}(k-1, j)$$

- Sinon, c'est que p ne passe pas par $k-1$ et donc que tous ses sommets intermédiaires sont dans $[\![0, k-2]\!]$, ie p est en fait dans $C^{k-1}_{i,j}$. Dans ce cas :

$$\mathscr{L}^k(i,j) = \mathscr{L}^{k-1}(i,j)$$

Comme l'une et une seules de ces possibilités se produit, cela donne la relation :

$$\mathscr{L}^k(i,j) = \min\left(\mathscr{L}^{k-1}(i, k-1) + \mathscr{L}^{k-1}(k-1, j), \mathscr{L}^{k-1}(i,j)\right)$$

On a donc une relation de Bellman. Pour l'initialisation, on a :

$$\mathscr{L}^0(i,j) = \begin{cases} +\infty & \text{si } i \text{ et } j \text{ ne sont pas reliés} \\ w_{i,j} & \text{sinon} \end{cases}$$

De plus, le travail ci-dessus permet de calculer la longueur minimale $\mathscr{L}^k(i,j)$ mais aussi le chemin minimal. Pour i (point de départ), j (point d'arrivée) et k (plus grande valeur de sommet intermédiaire) fixés, on a deux choix :

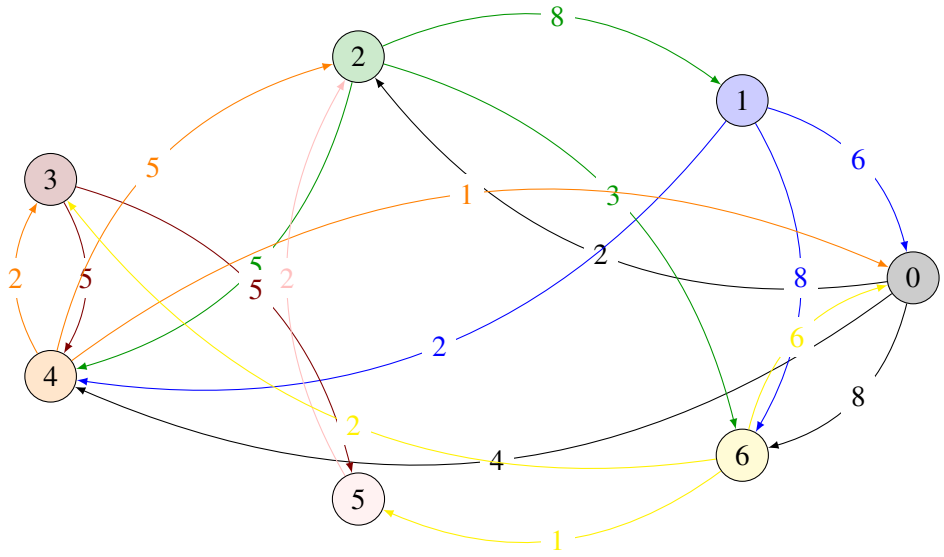
- Soit $\mathscr{L}^{k-1}(i,j) \leqslant \mathscr{L}^{k-1}(i, k-1) + \mathscr{L}^{k-1}(k-1, j)$, dans ce cas, le chemin optimal de $C^k_{i,j}$ ne passe pas par $k-1$, c'est déjà le chemin optimal qui permet de réaliser $\mathscr{L}^{k-1}(i,j)$.
- Sinon, c'est que le chemin optimal de $C^k_{i,j}$ passe par $k-1$, et ce chemin est alors la concaténation de p_1 et p_2, où p_1 est le chemin optimal de $C^{k-1}_{i,k-1}$ et p_2 le chemin optimal de $C^{k-1}_{k-1,j}$.

⋆ Exemple de donnée

Voici par exemple, une matrice d'adjacence et le graphe associé :

```
W = array(
    [[INF, INF, 2, INF, 4 , INF, 8],
     [6, INF, INF, INF, 2 , INF, 8],
     [INF, 8, INF, INF, 5 , INF, 3],
     [INF, INF, INF, INF, 5 , 5, INF],
     [1, INF, 5, 2, INF , INF, INF],
     [INF, INF, 2, INF, INF, INF , INF],
     [6, INF, INF, 2, INF, 1 , INF]
     ])
```

Le graphe associé est :



✯ Programmation en Python, méthode ascendante

On souhaite donc mettre en place cet algorithme par une programmation en Python. Dans un premier temps, on va plutôt utiliser une méthode itérative.

Pour la structure de données :

- Le graphe sera représenté par un `2d-array` `W`. Ainsi, `W[i,j]` sera la longueur de l'arrête (i,j), avec par convention que si il n'y a pas de telle arrête alors `W[i,j] = INF` où `INF` est une constante.
  Vous disposez de quelques exemples.
- un chemin $p = (i_0, \dots, i_l)$ sera représenté comme une liste.
- Pour calculer les longueurs optimales, on utilisera un `3d-array` `L`. Ainsi, `L[i,j,k]` sera la longueur d'un chemin optimal permettant de relier i à j en ne passant que par des sommets de $[\![0, k-1]\!]$. Pour calculer les chemin optimaux, on utilisera un dictionnaire `D` indexé sur des 3-uplets (i,j,k) ainsi, `D[(i,j,k)]` sera une liste p contenant l'un des chemins de longueur minimale permettant de relier i à j en ne passant que par des sommets de $[\![0, k-1]\!]$.
  NB : il faut utiliser des tuples et non des listes pour les indices.

R Dans ce td, on ne cherche pas à optimiser la mémoire. Ainsi, on garde en mémoire les longueurs et les chemins optimaux pour toutes les valeurs de k. En pratique, ces valeurs sont écrasées au fur et à mesure des itérations.

Le principe de l'algorithme est :

- d'initialiser les valeurs `L[i,j,0]` et `D[(i,j,0)]` pour tout (i,j). En utilisant :

$$\mathscr{L}^0(i,j) = \begin{cases} +\infty & \text{si } i \text{ et } j \text{ ne sont pas reliés} \\ w_{i,j} & \text{sinon} \end{cases}$$

  On pose donc : `L[i,j,0] = W[i,j]` et, si i et j sont liés alors `D[(i,j,0)] = [i,j]` sinon c'est une liste vide.
- Pour chaque k de 1 à n.
  On calcule `L(i,j,k)` et `D[(i,j,k)]` pour toutes les valeurs de i et j.
  On utilise la relation :

$$\mathscr{L}^k(i,j) = \min\left(\mathscr{L}^{k-1}(i,k-1) + \mathscr{L}^{k-1}(k-1,j), \mathscr{L}^{k-1}(i,j)\right)$$

  On compare les valeurs de `L(i,j,k-1)` et `L(i,k-1,k-1) + L(k-1,j,k-1)`.
  - si c'est `L(i, j, k-1)` qui est inférieur, alors c'est que pour relier i à j, on n'a pas intérêt à passer par $k-1$. Dans ce cas :

$$\mathscr{L}^k(i,j) = \mathscr{L}^{k-1}(i,j)$$

    et le chemin optimal est celui déjà calculé.
    Ainsi :

```
L[i, j, k] = L[i, j, k-1]
D[(i, j, k)] = D[(i, j, k-1)]
```

– Si c'est `L(i,k-1,k-1) + L(k-1,j,k-1)`, alors c'est que le chemin minimal de $C^k_{i,j}$ passe par $k-1$. Dans ce cas :

```
# somme de réels pour les longueurs:
L[i, j, k] = L[i, k-1, k-1] + L[k-1, j, k-1]
# concaténation de listes pour les chemins
D[(i, j, k)] = D[(i, k-1, k-1)][:-1] + [k-1] + D[(k-1, j, k-1)][1:]
```

⋆ Programmation en Python, méthode récursive

On va utiliser la mémoïsation.

On change la structure de données, on va donc utiliser un dictionnaire plutôt qu'une liste pour faciliter la recherche par clé.

Ainsi, `L` sera aussi un dictionnaire indexé sur les tuples (i,j,k) (comme `D`).

On écrit alors une fonction `RFWrec(W)` prenant en entrée la matrice W d'adjacence et calculant comme précédemment la liste des longueurs des chemins optimaux, et les chemins optimaux.

Cette fonction :

- initialise les dictionnaires `L` et `D`,
- contient une fonction récursive `calculeLD(i,j,k)`, prenant en entrée `k` et permettant via un appel récursif de remplir les dictionnaires. Le principe est simple :
  - Si la clé (i,j,k) existe dans `L` et `D`, on renvoie cette valeur.
  - Sinon, on la calcule via un appel récursif et on remplit les dictionnaires avec cette valeur.
- Enfin, la fonction `RFWrec(W)` appelle `calculeLD` pour $k = n$ et pour toutes les valeurs de (i,j).
- On renvoie ainsi les longueurs des chemins optimaux et les chemines optimaux.

Annexe : quelques données et résultats

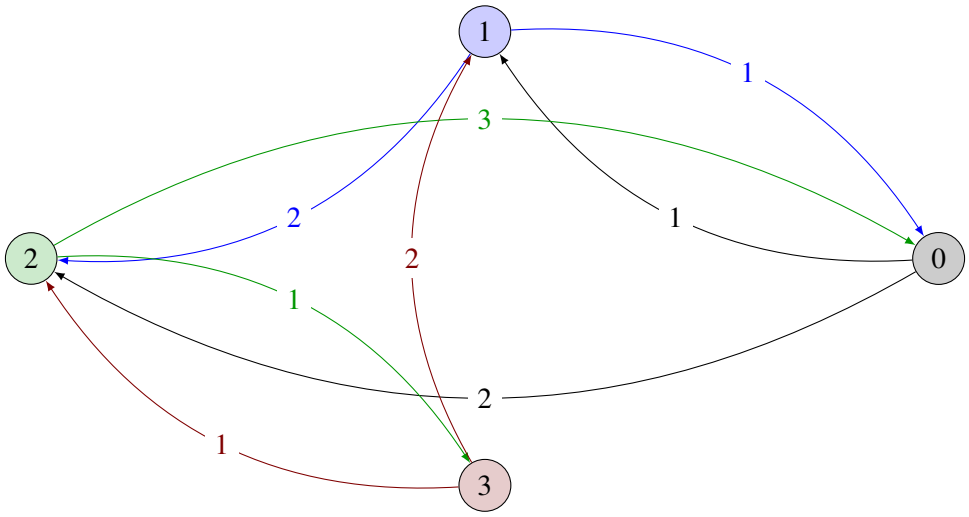
Pelletier Sylvain PSI, LMSC

✯ Premier test

Matrice d'adjacence :

```
W = array(
    [[INF, 1, 2, INF],
     [1, INF, 2, INF],
     [3, INF, INF,1 ],
     [INF, 2, 1,INF ]])
```

Graphe :



Distances minimales :

	0	1	2	3
0	2	1	2	3
1	1	2	2	3
2	3	3	2	1
3	3	2	1	2

Chemins minimaux :

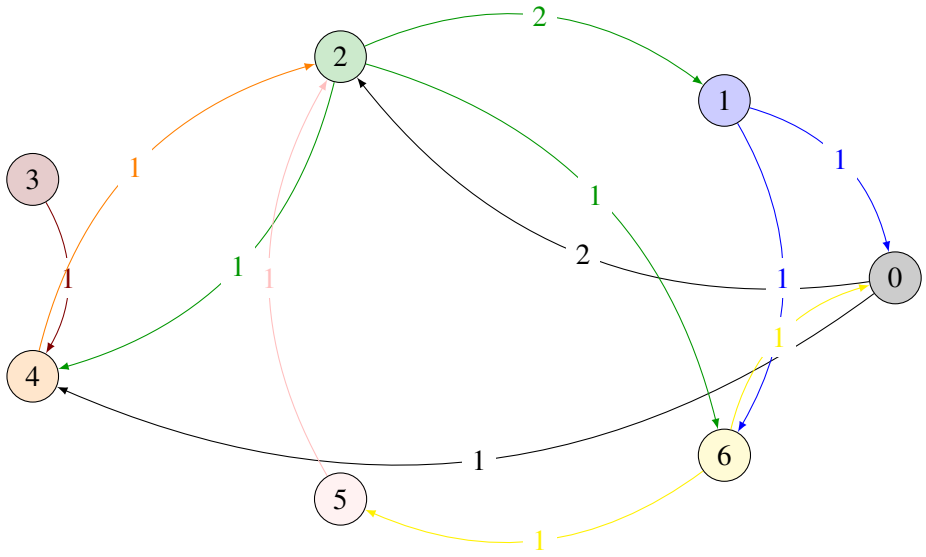
	0	1	2	3
0	[0, 1, 0]	[0, 1]	[0, 2]	[0, 2, 3]
1	[1, 0]	[1, 0, 1]	[1, 2]	[1, 2, 3]
2	[2, 0]	[2, 3, 1]	[2, 3, 2]	[2, 3]
3	[3, 1, 0]	[3, 1]	[3, 2]	[3, 2, 3]

✯ Deuxième test

Matrice d'adjacence :

```
W = array(
    [[INF, INF, 2, INF, 1  , INF, INF],
     [1, INF, INF, INF, INF , INF, 1],
     [INF, 2, INF, INF, 1  , INF, 1],
     [INF, INF, INF, INF, 1  , INF, INF],
     [INF, INF, 1, INF, INF , INF, INF],
     [INF, INF, 1, INF, INF, INF , INF],
     [1, INF, INF, INF, INF, 1  , INF]
     ])
```

Graphe :



Distances minimales :

	0	1	2	3	4	5	6
0	4	4	2	1000	1	4	3
1	1	5	3	1000	2	2	1
2	2	2	2	1000	1	2	1
3	4	4	2	1000	1	4	3
4	3	3	1	1000	2	3	2
5	3	3	1	1000	2	3	2
6	1	4	2	1000	2	1	3

Chemins minimaux :

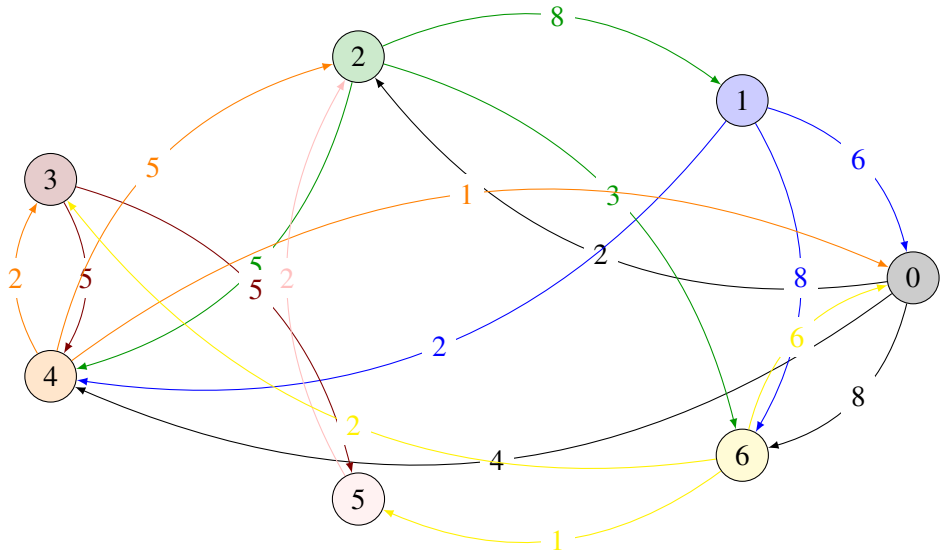
	0	1	2	3	4	5	6
0	[0, 4, 2, 6, 0]	[0, 4, 2, 1]	[0, 4, 2]	[]	[0, 4]	[0, 4, 2, 6, 5]	[0, 4, 2, 6]
1	[1, 0]	[1, 6, 5, 2, 1]	[1, 6, 5, 2]	[]	[1, 0, 4]	[1, 6, 5]	[1, 6]
2	[2, 6, 0]	[2, 1]	[2, 4, 2]	[]	[2, 4]	[2, 6, 5]	[2, 6]
3	[3, 4, 2, 6, 0]	[3, 4, 2, 1]	[3, 4, 2]	[]	[3, 4]	[3, 4, 2, 6, 5]	[3, 4, 2, 6]
4	[4, 2, 6, 0]	[4, 2, 1]	[4, 2]	[]	[4, 2, 4]	[4, 2, 6, 5]	[4, 2, 6]
5	[5, 2, 6, 0]	[5, 2, 1]	[5, 2]	[]	[5, 2, 4]	[5, 2, 6, 5]	[5, 2, 6]
6	[6, 0]	[6, 5, 2, 1]	[6, 5, 2]	[]	[6, 0, 4]	[6, 5]	[6, 5, 2, 6]

✯ Troisième test

Matrice d'adjacence :

```
W = array(
    [[INF, INF, 2, INF, 4 , INF, 8],
     [6, INF, INF, INF, 2 , INF, 8],
     [INF, 8, INF, INF, 5 , INF, 3],
     [INF, INF, INF, INF, 5 , 5, INF],
     [1, INF, 5, 2, INF , INF, INF],
     [INF, INF, 2, INF, INF, INF , INF],
     [6, INF, INF, 2, INF, 1 , INF]
     ])
```

Graphe :



Distances minimales :

	0	1	2	3	4	5	6
0	5	10	2	6	4	6	5
1	3	13	5	4	2	9	8
2	6	8	6	5	5	4	3
3	6	15	7	7	5	5	10
4	1	11	3	2	5	7	6
5	8	10	2	7	7	6	5
6	6	11	3	2	7	1	6

Chemins minimaux :

	0	1	2	3	4	5	6
0	`[0, 4, 0]`	`[0, 2, 1]`	`[0, 2]`	`[0, 4, 3]`	`[0, 4]`	`[0, 2, 6, 5]`	`[0, 2, 6]`
1	`[1, 4, 0]`	`[1, 4, 0, 2, 1]`	`[1, 4, 0, 2]`	`[1, 4, 3]`	`[1, 4]`	`[1, 4, 0, 2, 6, 5]`	`[1, 4, 0, 2, 6]`
2	`[2, 4, 0]`	`[2, 1]`	`[2, 6, 5, 2]`	`[2, 6, 3]`	`[2, 4]`	`[2, 6, 5]`	`[2, 6]`
3	`[3, 4, 0]`	`[3, 5, 2, 1]`	`[3, 5, 2]`	`[3, 4, 3]`	`[3, 4]`	`[3, 5]`	`[3, 5, 2, 6]`
4	`[4, 0]`	`[4, 0, 2, 1]`	`[4, 0, 2]`	`[4, 3]`	`[4, 0, 4]`	`[4, 0, 2, 6, 5]`	`[4, 0, 2, 6]`
5	`[5, 2, 4, 0]`	`[5, 2, 1]`	`[5, 2]`	`[5, 2, 6, 3]`	`[5, 2, 4]`	`[5, 2, 6, 5]`	`[5, 2, 6]`
6	`[6, 0]`	`[6, 5, 2, 1]`	`[6, 5, 2]`	`[6, 3]`	`[6, 3, 4]`	`[6, 5]`	`[6, 5, 2, 6]`

5 — Dictionnaires et table de hachage

I Dictionnaire en Python

Un dictionnaire ou tableau associatif est un conteneur dont les éléments sont des couples clé-valeur.

Avec comme propriété :

- L'association clé-valeur est une application, *ie* à une clé ne correspond qu'une valeur,
- la recherche d'un élément se fait sur la clé, *ie* l'instruction `in` est de la forme : `if cle in dic`.

Rappel : pour une liste, c'est la recherche par élément qui est favorisé, puisqu'on utilise `if elmt in liste`

Pour ces structures, on a les opérations suivantes :

- Création d'un structure vide. En python : `dic = {}`.
  On peut aussi créer des structures non vides. En donnant directement les couples clé-valeur :

```
dic = { cle1:valeur1, cle2: valeur2}
```

  Par compréhension :

```
dic = { cle:valeur for cle in conteneur if condition}
```

- Insertion d'un couple clé-valeur. En Python, l'instruction : `dic[cle] = valeur` ajoute le couple clé-valeur dans `dic` si celle-ci n'existe pas, modifie la valeur si la clé existe.
- La suppression d'une clé et de la valeur associée. En Python, l'instruction `del(dic[cle])` efface la clé.
- Le recherche d'une clé en temps O(1) en moyenne. Par l'instruction `cle in dic`.

Ainsi, pour un dictionnaire, c'est plutôt les clés qui sont importantes que les valeurs. En particulier, le temps de recherche d'une clé ne dépend pas de la taille du dictionnaire,

alors que pour une liste, il est linéaire en la taille. Cela montre que les dictionnaire et les clés sont stockées de manière très différentes en mémoire.

Le parcours d'un dictionnaire se fait aussi par les clés, on utilise ainsi :

```
for cle in dic
```

alors que pour les suites, `for x in L` boucle sur les éléments.

> Le but de ce chapitre est d'exposer le principe de fonctionnement bas niveau d'un dictionnaire.

On rappelle rapidement comment est stockée une liste en Python. (voir le chapitre révision pour plus de détail).

- La RAM est une suite de 0 et de 1 organisé de manière linéaire, on se repère dans la RAM par l'adresse mémoire. La RAM est inerte et ne peut pas modifier les valeurs. Si des valeurs doivent être déplacés, il faut les copier dans le processeur pour les écrire à un autre endroit ce qui est presque aussi coûteux en temps que de faire un calcul.
- Pour faire un calcul, le processeur va lire dans la RAM à un endroit donnée, des valeurs, fait des calculs et écrits dans la RAM.
- Le processeur a accès à tout endroit dans la RAM à la même vitesse (ce n'est pas une tête de lecture qui se déplace), il lui suffit d'avoir l'adresse mémoire.
- L'organisation de la RAM en ligne fait qu'il y est difficile de stocker des objets qui changent de taille : il est compliqué de réserver de la place contiguë dans la RAM, il est difficile de déplacer les éléments.

Pour stocker des conteneurs la première solution est celle des listes chaînées :

- On stocke l'adresse mémoire du premier élément, ainsi que la taille de la liste.
- à cette adresse, on stocke le premier élément, et l'adresse mémoire du deuxième élément,
- etc.

Ainsi, les éléments ne sont pas stockées dans des cases contiguës, mais répartis chacun à leur place dans la RAM. (Faire le schéma). Pour cette structure : il est facile d'insérer des éléments à n'importe quel endroit dans la liste, mais compliqué d'accéder à l'élément i. Par exemple, l'accès au dernier élément de L impose de parcourir toute la liste.

Lorsque l'on crée une liste, Python réserve en mémoire une place contiguë de taille fixée pour y stocker les adresses mémoires des éléments de la liste.

- Quelque soit la taille de la liste crée, il réserve en RAM une place contiguë pour quelques adresses mémoires.
- on y stocke les adresse mémoire des éléments de la liste. On ne stocke donc pas les éléments, mais leur adresse.
- Lorsque l'on veut accéder à l'élément $L[i]$, Python va lire son adresse mémoire. Pour cela, il connaît l'adresse mémoire de la liste L et ajoute i fois la faille d'une adresse. On comprend ainsi pourquoi les indices en Python commencent à 0.
- Si la taille de la liste dépasse le nombre de places prévues, Python modifie la taille en redemandant de la place contiguë et en déplaçant la liste.

C'est moins optimal pour la place en mémoire (puisque l'on utilise de la place contiguë) mais cela permet d'avoir accès à l'élément $L[i]$ directement.

L'ajout d'un élément à la fin est très facile, c'est plus compliqué d'insérer un élément au milieu (nécessite de déplacer des éléments).

Plus précisément, le temps nécessaire pour ajouter un élément à la fin de la liste n'est pas constant : lorsque l'ajout de l'élément fait que la liste dépasse la taille allouée en mémoire, on impose de créer une nouvelle liste. Hormis ce cas particulier, l'ajout d'un élément est rapide en fin de liste.

Enfin, si l'avantage de stocker les adresses mémoires et non les valeurs est que si un élément change de type (et donc de taille en mémoire), cela ne nécessite pas des opérations de recopie.

II Table de hachage

La structure de dictionnaire est créée et utilisé ainsi :

- On détermine l'ensemble $\mathscr{U}$ des **clés** envisageables.
  En Python, c'est tous les entiers, les flottants, les chaînes de caractères et les tuples, mais pas les listes (pas d'objet mutable).
  L'ensemble $\mathscr{U}$ est théoriquement fini mais son cardinal est très grand. On peut considérer qu'il est infini dénombrable.
- On se donne plusieurs fonctions de la forme

  $$h : \mathscr{U} \to \mathbb{N}$$

  chacune de ses fonctions prend ses valeurs dans un intervalle $[\![0, m_h - 1]\!]$. Ces fonctions sont les **fonctions de hachage**.
  Ainsi, pour une fonction h donnée et une clé c de $\mathscr{U}$, on a $h(c) \in [\![0, m_h - 1]\!]$.
  Bien entendu, h n'est pas injective et m_h est très petit devant $\text{Card}(\mathscr{U})$.
- Lorsque l'on crée un dictionnaire, l'une de ces fonctions h est choisie, et un tableau de taille m_h est réservée en mémoire.
  On appelle **alvéoles** les éléments de ce tableau.
- Pour insérer un couple clé-valeur, on calcule l'image $i = h(c)$ de la clé. On note $i = h(c)$, et on l'appelle la **valeur de hachage** de la clé c.
- Dans l'alvéole d'indice i (ie la case $h(c)$ du tableau), on stocke l'adresse mémoire d'une liste chaînée de couple clé, valeur.
  On ne peut pas stocker que la valeur car h n'étant pas injective, il y a parfois des **collisions** : deux clés c et c' ont la même valeur par h. Cette situation bien que rare est possible.
  Ainsi : si l'alvéole n'est pas encore utilisée, on crée une nouvelle liste chaînée dans laquelle on met le couple (clé,valeur), si la case est utilisée (ie si il y a collision), on ajoute à la fin de la liste chaînée le couple (clé,valeur).
  Cette structure est appelée **table de hachage**.
- On souhaite limiter le nombre de collision tout optimisant l'utilisation des alvéoles.
  On calcule donc le **taux de remplissage** (ou facteur de remplissage) : $\alpha = \frac{n}{m_h}$ où n est le nombre d'alvéole utilisée. Plus α est proche de 1, plus il y a de risque de collision. si α dépasse un certain seuil, on change de fonction de hachage.

Ainsi, l'instruction `cle in dic` fonctionne ainsi :

- on calcule la valeur $h(c)$ pour la clé c. On a donc accès à l'alvéole correspondante.

- Si l'alvéole est vide, c'est que la clé c n'est pas dans `dic`.
- Sinon, on a accès à une case mémoire qui contient la liste des couples clé valeur stockée en $h(c)$. Cette liste est généralement constituée d'un seul couple. Si il y a collision, elle peut contenir plusieurs valeurs, mais généralement très peu, il suffit donc de parcourir cette liste pour regarder si la clé est présente.

La complexité ne dépend donc pas de la taille, il suffit de calculer $h(c)$ (plus une petite recherche sur les listes chaînées si il y a eu collision).

III Exemple de fonctions de hachage

Les fonctions de hachage sont donc des fonctions de $\mathscr{U}$ dans $\mathbb{N}$, où $\mathscr{U}$ est l'ensemble des clés admissibles. Dans les exercices, on suppose que $\mathscr{U} = \mathbb{N}$. L'étude des fonctions de hachage amène souvent à des exercices de mathématiques : en considérant que les clés sont des entiers répartis aléatoirement selon telle probabilité, on cherche à calculer la probabilité qu'il y ait collision.

Pour mesurer la qualité d'un hachage, on doit vérifier deux points :

- Les valeurs de $h(c)$ pour $c \in \mathbb{N}$ doivent être le plus possible équi-réparties (pas de points qui revient plus souvent que les autres). Cela permet d'avoir un taux de remplissage le plus constant des alvéoles.
- si $h(a) = h(b)$ alors a et b doivent être très différents.

Le hachage a d'autres applications que le stockage des dictionnaires. Par exemple, cela sert à détecter des erreurs comme pour le `md5sum` cela permet aussi de faire de la cryptographie.

III.1 Division euclidienne

La fonction de hachage la plus simple est le reste de la division euclidienne : on considère une entier m et on utilise

$$h : c \longmapsto c\%m \text{ reste de la division euclidienne de } c \text{ par } m$$

La taille de hachage de cette fonction est m puisqu'elle prend ses valeurs dans $[\![0, m-1]\!]$.

Par exemple, si on sait que les clés sont des entiers uniformément répartis dans $[\![0, N-1]\!]$, et que l'on veut placer k clé on note $(X_i)_{i\in[\![1,k]\!]}$ les VAR donnant les clés, et on cherche la probabilité que la clé i et la clé j soient en collision. Ceci s'écrit :

$$A_{i,j} = (X_i = X_j) = (X_i \equiv X_j[m])$$

et on cherche $p(A_{i,j})$. Cela s'écrit :

$$\begin{aligned} p(A_{i,j}) =& p(m|X_i - X_j) \\ =& \sum_{k\in\mathbb{Z}} (X_i = X_j + km) = \dots. \end{aligned}$$

ceci amène à des exercices de mathématiques.

III.2 Multiplication

Une autre méthode pour construire des fonctions de hachage consiste à se donner un réel $\in\in]0,1[$ et un entier $m > 0$. On définit :

$$h : c \longmapsto \lfloor m \times (c \times \theta \text{mod} 1) \rfloor$$

autrement dit : on calcule d'abords : $c \times \theta \text{mod} 1$, c'est à dire :

$$a = c \times \theta - \lfloor c \times \theta \rfloor \text{ ie on garde la partie décimale}$$

puis on calcule $\lfloor a \times m \rfloor$.

La taille de hachage de cette fonction est m puisqu'elle prend ses valeurs dans $[\![0, m-1]\!]$.

Voici la fonction que l'on peut écrire :

```
def hachage(m, theta, c):
    a = c*theta - int( c*theta )
    return int( a*m )
```

Il faut choisir θ de telle manière que les clés soient répartis le plus uniformément possible. Des études théoriques assurent que l'on obtient le meilleur résultat avec $\theta = \frac{\sqrt{5}-1}{2}$.

Pour illustrer, on va hacher les entiers de $[\![0, (10^5 - 1)]\!]$ avec cette méthode en prenant $\theta = \frac{\sqrt{5}-1}{2}$ et $m = 101$.

```
m = 101
theta = (sqrt(5) -1)/2
hach = { i:[] for i in range(m)}
for c in range(10**5):
    h = hachage(m, theta, c)
    hach[h].append(c)
for i in range(m):
    plot(i, len(hach[i]), "o")
show()
```

On constate que tous les nombres $i \in [\![0, 100]\!]$ ont presque le même nombre d'antécédents par h.

On constate aussi que des valeurs proches ne peuvent être hachées à l'identique : si $h(a) = h(b)$, on a $|a - b| \geqslant 55$, donc deux clés proches ne sont jamais hachées à l'identique.

```
for i in range(m):
    listeDiff = [ (hach[i][j+1] - hach[i][j])
                  for j  in range(len(hach[i])-1)]
    diffmin = min (listeDiff)
    print("diférence minimale entre deux antécédants de",
                              i, ": ", diffmin)
    plot(i, diffmin, "o")
```

6 — Algorithme pour l'étude des jeux

I Vocabulaire de la théorie des jeux

Il s'agit essentiellement de formaliser le vocabulaire appris avec le TD « jeu de Nims ».

On considère un jeu à deux joueurs (noté J_0 et J_1), pour lequel :
- l'issue du jeu peut être la victoire de J_0, la victoire de J_1 ou la partie nulle.
- Chaque joueur joue à son tour en modifiant L'ÉTAT DU JEU.
- Il n'y a pas de hasard.
- Les joueurs ont accès à la même information (pas de cartes cachées).

On représente alors le jeu sous la forme d'un *arbre de jeu*. C'est un arbre, ie un **graphe orienté, connexe et sans cycle**. La **racine** est la situation initiale. Une **branche** est une séquence de coups dans une partie. Les **noeuds** de l'arbre sont les positions dans le jeu, les **arrêtes** représentent les mouvements.

■ **Exemple I.1** L'exemple que l'on va suivre et que l'on a déjà vu en TD est celui du jeu de Nims.

Dans sa version la plus simple, on part d'un tas de N allumettes et chaque joueur peut prendre une, deux ou trois allumettes.

L'état du jeu est donc le couple : nombre d'allumettes, parité du numéro du tour (pour savoir si c'est le joueur 1 ou le joueur 2 qui joue). ■

Un état où le joueur 0 a le trait (ie c'est son tour de jouer) est dit **contrôlé** par J_0, de même un état où le joueur 2 a le trait est dit contrôlé par J_1.

On parle de **graphes bipartis** puisque chaque état est contrôlé par le joueur 0 ou le joueur 1.

✯ Définitions sur les graphes

Définition I.1 Un graphe orienté G est la donnée d'un ensemble fini S (ensemble des **sommets**) et d'une partie de S^2 (l'ensemble des **arc**). On note $G = (S, A)$.

Pour un couple $(x,y) \in A$, x est **l'origine** de l'arc, et y est **l'extrémité** de l'arc.
Le graphe n'est pas **orienté** si :

$$\forall (x,y) \in S^2, (y,x) \in A \iff (x,y) \in A$$

(autrement dit si les arc sont toutes à double sens). Dans ce cas, on parle plutôt d'arêtes que d'arc.

Dans le cas contraire, on dit donc que le graphe est orienté.

Pour tout $s \in S$, les successeurs de s sont les extrémités des arcs dont s est l'origine.

Les prédécesseurs de s sont les origines des arcs dont s est l'extrémité.

Un sommet est terminal si il n'a pas de successeur.

✯ Jeux et graphes

Définition I.2 Le graphe $G = (S,A)$ est biparti si il existe deux sous-ensembles de sommets S_0 et S_1, formant une partition des sommets S, tel que pour tout arrête $(x,y) \in A$, l'origine x et l'extrémité y de l'arc ne sont pas dans la même partie (ie $x \in S_0 \iff y \in S_1$).

Un **graphe de jeu à deux joueurs** ou **arène** est une structure (G,S_0,S_1) où $G = (S,A)$ est un graphe fini et biparti. S_0 et S_1 sont la partition de S comme ci-dessus.

On définit aussi les **conditions de gains** pour le joueur J_0 : c'est un sous-ensembles des sommets terminaux où J_0 a gagné, et idem pour le joueur J_1.

■ **Exemple I.2** Jeu de nims avec 3 allumettes : le joueur 0 joue en premier, celui qui prend la dernière a perdu.

Les états du jeu sont des couples (k,j) avec $k \in [\![0,3]\!]$ le nombre d'allumettes et $j \in [\![0,1]\!]$, le joueur qui a le trait.

On a alors un graphe biparti. (à dessiner).

Les états sont :

$$(3,0) \to (2,1), (1,1), (0,1)T(2,1) \to (1,0), (0,0)T(1,1) \to (0,0)T(1,0) \to (0,1)T$$

■

Pour résumer, la donnée du jeu correspond :

- À la donnée du graphe biparti (S,A), S est l'ensemble des positions, A l'ensemble des coups possibles. Chaque coup relie une position de S_0 (ie où le joueur 0 a le trait) vers une position de S_1 (où le joueur 1 a le trait) ou l'inverse.
- une position de départ (qui est un sommet de S_0).
- des conditions de gains pour les deux joueurs.

Avec ce vocabulaire, un coup possible est donc une arrête qui relie deux sommets. Lorsqu'un joueur joue, le jeu est à un des sommets s du graphe (contrôlé par J_0 ou J_1 qui a le trait). Il y a deux possibilités :

- si s est terminal alors le joueur ne peut plus jouer, la partie est finie (et donc il peut avoir perdu, gagné ou partie nulle).
- il choisit un arc (s,s') d'origine s dans la graphe. s' est contrôlé par l'autre joueur.

✯ Parties et stratégies

Définition I.3 Un chemin dans la graphe orienté G est une suite (finie ou non) de sommets $(s_i)_{i\in I}$ reliés par des arcs, ie :

$$\forall i \in I,\ (s_i, s_{i+1}) \in A$$

Un chemin fini est maximal lorsque l'extrémité de son dernier arc est terminale.
Dans un graphe de jeu, une partie finie débutant en s_0 (position de départ) est un chemin fini maximal de G dont le premier sommet est s_0. Une partie partielle débutant en s_0 est un chemin fini dont le premier sommet est s_0.

Définition I.4 Dans un graphe de jeu $((S,A),S_0,S_1)$:

- Une stratégie (sans mémoire) pour le joueur J_0 est une application :

$$\sigma : \begin{cases} S_0 & \to & S_1 \\ x & \longmapsto & y \end{cases}$$

  qui à une position x du jeu où le joueur 0 doit jouer, associe un sommet y (le choix d'un coup), avec $(x,y) \in A$ (ie le coup est jouable).
  Une stratégie permet ainsi de savoir quel coup jouer (de manière unique car σ est une application) étant donné une situation de jeu.
- Une partie $(s_0,\dots,s_{2t})$ est conforme à une stratégie pour le joueur 0 si :

$$s_2 = \sigma(s_1),\ s_4 = \sigma(s_3),$$

  (autrement dit, le joueur J_0 a toujours utilisé l'application σ pour choisir son coup).
- On définit de même la notion de stratégie sans mémoire pour le joueur J_1 et la notion de partie conforme à une stratégie pour le joueur 1.
- Une fois choisir une stratégie pour le joueur 0, une position $d \in S$ est gagnante pour le joueur J_0 lorsque toute partie conforme à la stratégie et qui débute en d est gagné par J_0 (ie finit par passer par un sommet qui vérifie la condition de gain pour le joueur 0).
  Une stratégie est gagnante pour le joueur J_0 si toute partie conforme à la stratégie qui commence par la position de départ est gagnante pour J_0

R On se restreint aux stratégies sans mémoire : il suffit d'observer la position sur le jeu pour décider quel coup jouer (sans avoir besoin de l'historique des coups précédents).

■ **Exemple I.3** Pour le jeu de Nims 1d avec 3 allumettes, une stratégie consiste à écrire :

$$(3,0) \to (1,1)(1,0) \to (0,1)$$

avec les notations précédentes. ■

■ **Exemple I.4** En TD, on a vu le jeu de nims en 2d. On a construit une stratégie : si une position est gagnants, jouer le coup gagnant, si une position est perdante, jouer un coup au hasard. ■

II Calcul des attracteurs dans les jeux d'accessibilité

Définition II.1 Un jeu est dit d'accessibilité lorsque :
- Les conditions de gains sont des sommets terminaux (toujours le cas dans notre modèle),
- il n'y a pas de match nul (pour simplifier).

On note F_0 l'ensemble des sommets terminaux où le joueur 0 gagne, et F_1 l'ensemble des sommets terminaux où le joueur 1 gagne. Il ne peut pas y avoir d'autres sommets terminaux.

Dans un jeu d'accessibilité, lorsque c'est au joueur J_0 de jouer il est sûr de gagner lorsque :
- Il est au trait et il peut choisir un coup qui le mène à une position gagnante.
- L'adversaire est au trait et tous ses coups jouables mènent à une position gagnante pour le joueur 0.

On cherche à calculer une partie des sommets notée $\mathscr{G}$ tels que si la position du jeu fait partie de A, le joueur 0 est sûr de gagner.

Supposons que l'on dispose d'une partie des sommets notée $\mathscr{G}_n$ telle que tous les sommets de $\mathscr{G}_n$ soit gagnant pour le joueur 0 en moins de n coups (ie le joueur 0 joue au plus n fois).

On peut alors définir les ensembles suivants :

$$\left\{ s \in S_0 \middle| \exists (s,s') \in A, s' \in \mathscr{G}_n \right\}$$

c'est l'ensemble des positions (sommets) contrôlés par le joueur 0 qui permettent au joueur 0 d'amener le jeu en une position gagnante (ie de $\mathscr{G}_n$). Autrement dit qui permettent de forcer le joueur 1 à recevoir une position perdante pour lui.

On a aussi :

$$\left\{ s \in S_1 \middle| \forall (s,s') \in A, s' \in \mathscr{G}_n \right\}$$

c'est l'ensemble des positions (sommets) contrôlés par le joueur 1 qui l'oblige à amener le jeu en une position de $\mathscr{G}_n$.

On voit que cela permet de construire $\mathscr{G}_{n+1}$ en ajoutant à $\mathscr{G}_n$ les deux ensembles précédents.

Puis il faudra prendre la réunion de tous les $\mathscr{G}_n$ (ensemble des états gagnants en n coups) pour obtenir $\mathscr{G}$ (ensemble des états gagnants en un nombre quelconque de coups).

À partir de cette idée, on construit une suite de parties des sommets croissantes, on pose pour l'initialisation :

$$\mathscr{G}_0 = F_0 \text{ ensemble des positions gagnées pour le joueur 0}$$

puis on itère pour $n \in \mathbb{N}$:

$$\mathscr{G}_{n+1} = \mathscr{G}_n \cup \left\{ s \in S_0 \middle| \exists (s,s') \in A, s' \in \mathscr{G}_n \right\} \cup \left\{ s \in S_1 \middle| \forall (s,s') \in A, s' \in \mathscr{G}_n \right\}$$

On constate que $\mathscr{G}_n$ est l'ensemble des positions gagnantes pour le joueur 0 (que

ce soit à lui de jouer ou à l'adversaire. Plus précisément lorsque $s \in \mathscr{G}_n$, on peut dire que le joueur 0 gagne en moins de n coups.

La suite $(\mathscr{G}_n)$ étant croissante pour l'inclusion, et l'ensemble S des sommets étant finis, elle est nécessairement stationnaire : il existe un rang n_0 tel que $\forall n \geqslant n_0, \mathscr{G}_n = \mathscr{G}_{n_0}$.

On note alors $Attr(F_0, J_0) = \mathscr{G}_{n_0}$.

Définition II.2 L'ensemble $Attr(F_0, J_0)$ est le bassin d'attraction de F_0 pour le joueur J_0 : c'est l'ensemble des positions à partir desquelles, la joueur J_0 est sûr d'arriver en une position gagnante de F_0. Autrement dit l'ensemble des positions gagnantes.

Notons qu'en l'absence de position nulle, on a $S \setminus Attr(F_0, J_0)$ est l'ensemble des positions gagnantes pur le joueur 1.

Pour tout $s \in Attr(F_0, J_0)$ on définit le rang de s :

$$rg(s) = \min\{i | s \in A_i\}$$

C'est le nombre de coup minimal pour gagner à partir de s.

Lorsque $s \notin Attr(F_0, J_0)$, on pose par convention : $rg(s) = +\infty$.

À partir de ces définitions, on peut proposer un algorithme de calcul des positions gagnantes pour le joueur 0 :

- On part des positions gagnantes F_0. C'est l'ensemble $\mathscr{G}_0$.
- Pour un n donné, on construit A_{n+1} à partir de $\mathscr{G}_n$:
  - On construit l'ensemble des positions s contrôlées par le joueur 0 telles qu'il existe une arrête partant de s et allant vers une position de $\mathscr{G}_n$
  - On construit l'ensemble des positions s contrôlées par le joueur 1 telles que quelque soit l'arrête partant de s, on arrive dans une position de $\mathscr{G}_n$.
  - L'ensemble $\mathscr{G}_{n+1}$ est alors la réunion de $\mathscr{G}_n$ et des deux ensembles précédents.
- On continue ainsi jusqu'à convergence.

III Algorithme de Min-Max

On considère donc une position donnée dans un jeu et l'on cherche à répondre à la question suivante : quel est le meilleur coup à jouer ? Le calcul des attracteurs de la section précédente n'est pas toujours possible, il est souvent trop long. On se déplace plutôt dans le cadre où l'on va regarder l'ensemble des parties comme un arbre et chercher la meilleure branche en calculant un nombre donné de coups à l'avance.

⋆ Arbres

On commence par définir la structure qui nous servira pour ce problème :

Un arbre est un graphe orienté dans lequel :
- Il existe un sommet et un seul appelé **racine** n'ayant pas d'antécédent.
- Tous les autres sommets admettent un prédécesseur et un seul.

De manière pus formelle, un arbre est un ensemble $\mathscr{A}$ non vide dont les éléments sont appelés **noeuds** et sur lequel est défini une relation $\mathscr{P}$ avec : $x\mathscr{P}y$ signifie x est père de y tel que :
- il existe un noeud r et un seul appelé **racine** n'ayant pas de père.
- tous les autres éléments admettent un père et un seul.

La propriété fondamentale des arbres est que quelque soit le noeud, il existe un chemin qui permet de le relier à la racine. Pour tout $x \in \mathscr{A} \setminus \{r\}$, il existe une suite de noeud

$$x_0 = r, \ldots, x_p = x$$

tel que pour tout $i \in [\![0, p-1]\!]$, on ait $x_i \mathscr{P} x_{i+1}$.

on a le vocabulaire suivant :
- les sommets d'un arbre sont appelés **noeuds**,
- les noeuds sans descendants sont les **feuilles** de l'arbre,
- le **degré** d'un noeud est le nombre de descendants,
- la **hauteur** d'un arbre est la profondeux
- Un arbre est dit **étiqueté** lorsqu'à chaque noeud on associe une information appelée étiquette.

Dans le cas qui nous intéresse, la situation de départ est donc la racine de notre arbre. chaque branche correspond à un coup possible qui aboutit donc à d'autres noeuds étiquetés par une autre position. Les feuilles de l'arbre sont les fins de parties possibles (avec donc victoire d'un joueur ou de l'autre).

III.1 Algorithme du minmax

On considère que l'on dispose d'une fonction U qui évalue une position du point de vue du joueur 0. Cette fonction est positive si la position est favorable au joueur 0, négative sinon.

On parle de fonction d'évaluation ou fonction d'utilité.

■ **Exemple III.1** Pour le jeu de puissance4, on peut imaginer une fonction qui renvoie 1 si je joueur 0, -1 si il perd 0 si aucun joueur a gagné.

On peut aussi imaginer évaluer différemment : compter combien de pions de la même couleur sont alignés dans chaque ligne et chaque colonne.

■

Pour le joueur 0, une stratégie simple consiste à regarder tous les coups possibles et à choisir celui qui maximise cette fonction d'utilité.

Étant donné une position s, le meilleur coup t à choisir est celui qui réalise :

$$\max_{t \in succ(s)} U(t)$$

(il n'y bien sûr a pas unicité d'un tel coup). dans cette notation $succ(s)$ désigne les successeurs de s cad les noeuds reliés à s.

Si on va un peu plus loin, une stratégie un peu plus avancé consiste à choisir le coup qui réalise la plus grand utilité quelque soit la réponse de l'adversaire. Il faut alors trouver un coup t qui réalise :

$$\max_{t\in succ(s)} \begin{cases} \min_{u\in succ(t)} U(t) & \text{si } succ(t)\neq\emptyset \\ U(t) & \text{si } succ(t)=\emptyset \end{cases}$$

En prévoyant 3 coups en avance, cela donnerait (en négligeant les cas terminaux) :

$$\max_{t\in succ(s)} \min_{u\in succ(t)} \min_{v\in succ(u)} U(v)$$

Cette construction se généralise et porte le nom d'algorithme de min-max.

Soit U une fonction d'utilité pour le joueur 0, c'est-à-dire une fonction qui évalue les sommets (ie les positions) du point de vue du joueur 0.

On note S_0 l'ensemble des sommets où c'est le joueur 0 qui a le trait, et S_1 l'ensemble des sommets où c'est le joueur 1 qui a le trait.

On définit par récurrence pour $n \geqslant 0$ les **utilités maximales à n coups** noté $(U^r)_{r\in[\![0,n]\!]}$ par :

$$\forall s\in S,\ U^0(s)=U(s)$$

$$\forall n\geqslant 1,\ \forall s\in S,\ \begin{cases} U^{n-1}(s) & \text{si } succ(s)=\emptyset \\ \max_{t\in succ(s)} U^{n-1}(t) & \text{si } s\in S_0 \\ \min_{t\in succ(s)} U^{n-1}(t) & \text{si } s\in S_1 \end{cases}$$

Une stratégie pour le joueur 0 qui **réalise l'utilité maximale** en p coups consiste à choisir pour tout sommet $s\in S_0$ non terminal :

$$f(s)\in \text{argmax} U^p(t)$$

autrement dit qui consiste à choisir pour le joueur 0 une valeur qui réalise le maximum de l'utilité.

R L'algorithme de min max est assez simple à mettre en oeuvre, le problème principal est dans la fonction d'utilité. Il est assez difficile d'estimer une position.

On parle **méthodes d'heuristiques** pour désigner les méthodes qui fournissent des solutions acceptables sans que l'on ne puisse prouver leur optimalité.

Dans le contexte de l'algorithme de min max, on désigne par heuristique les fonctions d'évaluation que l'on utilise sans pouvoir prouver qu'elles permettent de gagner à coup sûr.

■ **Exemple III.2** Pour le puissance 4, une heuristique pou le joueur X consisterait à associer à une position :

- $+\infty$ si le joueur X a gagné, $-\infty$ sinon.
- à compter 2 points à chaque fois qu'il y a un alignement de 3X suivi d'une case vide, -2 points pour chaque alignement de 3 O suivi d'une case vide.
- de la même manière 1 point pour chaque alignement de 2X suivi de deux cases vides, etc.

- 0 pour les autres positions.

Une fois choisie une telle heuristique, une méthode min max a 3 pas est facile à mettre en oeuvre. ■

Annexe programme minmax pour le puissance 4

Pelletier Sylvain PSI, LMSC

Le programme suivant présent un minmax pour le jeu de puissance 4. Le jeu est représenté par une liste de 7 chaînes de caractères contenant des `'X'` ou des `'O'` et correspondant au contenu des colonnes.

La fonction `list2array` permet de transformer cette liste en un 2d-array de taille 6,7 correspondant au jeu.

On commence par définir une fonction d'utilité qui évaluer qui renvoie 1 si `'X'` a gagné, -1 si il a perdu et 0 si il n'y a pas de vainqueur.

On définit alors une première fonction `minmax1` qui permet de trouver le meilleur coup pour `'X'` pour une position donné simplement en choisissant le coup qui la plus forte utilité sans prendre en compte la réponse de l'adversaire. On définit ensuite une deuxième fonction `minmax2` qui prend en compte la réponse de l'adversaire.

Enfin, on définit une fonction `strategie` permettant de prendre en compte un nombre quelconque n de coups. Cette fonction se base sur le fonction récursive `utiliteMaximale` qui calcule l'utilité maximale à n coups.

```
from numpy import array
def prettyprint(jeu):
    """
    entrée: jeu = array (6,7) de str "X", " " ou "O"
                = situation de jeu
    sortie: rien affichage à l'écran
    """
    print("---- situation de jeu ----")
    msg = " 0 1 2 3 4 5 6\n"
    for i in range(6):
        msg += "|"
        for j in range(7):
            if jeu[i,j] !='':
                msg += jeu[i,j]+"|"
            else:
                msg += " |"
        msg += "\n"

    msg += " 0 1 2 3 4 5 6\n"
    print(msg)


def list2array(listeJeu):
    """
    entrée: listeJeu = liste de 7 de str "X" ou "O"
            contenu de chaque  colonne
    sortie: jeu = array (6,7) de str "X", " " ou "O"
                = situation de jeu
    """
    assert len(listeJeu) == 7
    jeu = array([[" " for i in range(7)] for j in range(6)])
    assert jeu.shape == (6,7)
    for j in range(7):
        for k in range(len(listeJeu[j])):
            i = 5-k
            jeu[i,j]= listeJeu[j][k]
    return jeu


def utilite(listeJeu):
    """
    entrée: listeJeu = liste de 7 de str "X" ou "O"
            contenu de chaque  colonne
    sortie 0 si pas de victoire
           1 (resp -1) si victoire "X" (resp "O")
    évalue le jeu pour X
    """
```

```
48      jeu = list2array(listeJeu)
50      assert jeu.shape == (6,7)
        horizontal = [ ([i,j], [i+1,j], [i+2,j], [i+3,j]) for i in range(3) for j in range(7) ]
52      vertical = [( [i,j], [i,j+1], [i,j+2], [i,j+3]) for i in range(6) for j in range(4) ]
        diagonaleNE = [ ([i,j], [i+1, j-1], [i+2, j-2], [i+3, j-3]) for i in range(3) for j in
54      diagonaleSE = [ ( [i,j], [i-1, j-1], [i-2, j-2], [i-3, j-3]) for i in range(3, 6)
for j in range(3,7) ]
        for extrait in horizontal + vertical +diagonaleNE + diagonaleSE:
56          liste4 = [jeu[i,j] for i,j in extrait]
            assert len(liste4) == 4
58          if liste4.count("X") == 4:
                #print(extrait, liste4)
60              return 1
            elif liste4.count("O") == 4:
62              #print(extrait, liste4)
                return -1
64      return 0

66  def minmax1(listeJeu):
        """
68      entrée: listeJeu = liste de 7 de str "X" ou "O"
            contenu de chaque  colonne
70      sortie: meilleur coup en choisissant 1 coup à l'avance
        """
72      print("---- minmax niveau 1 ---- ")
        listeCoupEval = {} # dictionnaire de la forme coupjoué / évaluation.
74      for i in range(7):
            if len(listeJeu[i]) <6 :
76              copieJeu = listeJeu.copy()
                copieJeu[i] += 'X'
78              listeCoupEval[i] = utilite(copieJeu)

80      # si besoin d'affichage:
        for i in listeCoupEval:
82          print("si X joue en ", i, "eval = ", listeCoupEval[i])
        # On cherche alors le max du jeu
84      valmax = -2
        indmax = 0
86      for i in listeCoupEval:
            if listeCoupEval[i] > valmax:
88              valmax = listeCoupEval[i]
                indmax = i
90
        return indmax
92
94  def minmax2(listeJeu):
        """
96      entrée: listeJeu = liste de 7 de str "X" ou "O"
            contenu de chaque  colonne
98      sortie: meilleur coup en choisissant 2 coup à l'avance
        """
100     print("---- minmax niveau 2 ---- ")
        listeCoupEval = {} # dictionnaire de la forme [i,j] / évaluation.
102     for i in range(7):
            if len(listeJeu[i]) <6 :
                copieJeu = listeJeu.copy()
104             copieJeu[i] += 'X'
                for j in range(7):
106                 if len(copieJeu[j]) <6 :
                        copieJeu2 = copieJeu.copy()
108                     copieJeu2[j] += 'O'
                        listeCoupEval[(i,j)] = utilite(copieJeu2)
110
```

```
    # si besoin d'affichage:
    for i,j in listeCoupEval:
        print("si X joue en ", i, "puis que O joue en ", j, "eval = ", listeCoupEval[i,j])

    listMin = {} # donne pour chaque i jouable le min listeCoupEval[i,j]
    for i,j in  listeCoupEval:
        if i not in listMin or listeCoupEval[i,j] < listMin[i] :
            listMin[i] = listeCoupEval[i,j]

    # si besoin d'affichage:
    for i in listMin:
        print("évaliuation en 2 coup si X joue en ", i, "min de l'éval = ", listMin[i])

    # On cherche alors le max de listMin
    valmax = -2
    indmax = 0
    for i in listMin:
        if listMin[i] > valmax:
            valmax = listMin[i]
            indmax = i

    return indmax

def utiliteMaximale(listeJeu, n, tour):
    """
    entrée: listeJeu = liste de 7 de str "X" ou "O"
        contenu de chaque  colonne
            n = int = niveau
            niveau 0 = fct utilite
            niveau 1 = max de fct utilite
            niveau 2 = minmax de fct utilite, etc.
        tour = "X" ou "O": joueur qui doit jouer
    sortie: évaluation de listeJeu à n niveau
    fct récursive
    """

    if n == 0 :
        # print("valeur sortie:", utilite(listeJeu))
        return utilite(listeJeu)

    symInv = "X" if tour == "O" else "O"

    evalSucc = {}
    for i in range(7):
        if len(listeJeu[i]) <6 :
            copieJeu = listeJeu.copy()
            copieJeu[i] += tour
            evalSucc[i] = utiliteMaximale(copieJeu, n-1, symInv)



    if len(evalSucc) == 0 :
        # pas de coup possible
        return utiliteMaximale(listeJeu, n-1, symInv)


    valmax = -2
    valmin = 2
    for i in evalSucc :
        if evalSucc[i] > valmax:
            valmax = evalSucc[i]
        if evalSucc[i] < valmin:
            valmin = evalSucc[i]

```

```
    if tour == "X":
176         return valmax
    else :
178         return valmin


180


182 def strategie(listeJeu, n):
    """
184     entrée: listeJeu = liste de 7 de str "X" ou "O"
            contenu de chaque  colonne
186             n = int = niveau
                niveau 0 = fct utilite
188             niveau 1 = max de fct utilite
                niveau 2 = minmax de fct utilite, etc.
190     sortie: meilleur coup en choisissant n coups à l'avance
    """
192     print("---- minmax niveau ", n," ---- ")
    listeCoupEval = {} # dictionnaire de la forme coupjoué / évaluation.
194     for i in range(7):
            if len(listeJeu[i]) <6 :
196             copieJeu = listeJeu.copy()
                copieJeu[i] += "X"
198             listeCoupEval[i] = utiliteMaximale(copieJeu, n, "O")

200     # si besoin d'affichage:
    for i in listeCoupEval:
202         print("si X joue en ", i, "éval après ",n, "coups: ", listeCoupEval[i])

204     # On cherche alors le max
    valmax = -1
206     indmax = 0
    for i in listeCoupEval:
208         if listeCoupEval[i] > valmax:
                valmax = listeCoupEval[i]
210             indmax = i
    print("évaluation finale de la position", valmax)
212     return indmax


214


216


218


220 print("--- début du programme IA pour  puissance4 ---------------")
listeJeu = ['', '', '', 'OXO', 'OXO', 'XO', 'OXXO']
222 print(utilite(listeJeu))


224
print("--- TEST 1  ---------------")
226 listeJeu = ['', '','' , 'XOX', 'XOX', 'OX','XO']
jeu = list2array(listeJeu)
228 prettyprint(jeu)
input("---Test minmax niveau 1 ---")
230 print("meilleur coup à jouer (1 coup à l'avance):", minmax1(listeJeu))


232
input("--- TEST 2  ---------------")
234
listeJeu = ['', '','' , 'OXO', 'OXO', 'XO','OX']
236 jeu = list2array(listeJeu)
prettyprint(jeu)
238 input("---Test minmax niveau 1  et 2 ---")
```

```
print("meilleur coup à jouer (1 coup à l'avance):", minmax1(listeJeu))
input("------")
print("meilleur coup à jouer (2 coups à l'avance):", minmax2(listeJeu))


input("--- TEST 3  ---------------")
listeJeu = ['', '','' , 'X', 'X', '','']
#listeJeu = ['', '','' , 'OXO', 'OXO', 'XO','OX']
jeu = list2array(listeJeu)
prettyprint(jeu)
input("---Test minmax différents niveaux ---")
print("meilleur coup à jouer (1 coup à l'avance):", strategie(listeJeu, 0))
input("------")
print("meilleur coup à jouer (2 coups à l'avance):", strategie(listeJeu, 1))
input("------")
print("meilleur coup à jouer (3 coups à l'avance):", strategie(listeJeu, 2))
input("------")


input("--- TEST 4  ---------------")
listeJeu = ['', '','' , 'O', 'O', '','']
jeu = list2array(listeJeu)
prettyprint(jeu)
print("meilleur coup à jouer (3 coups à l'avance):", strategie(listeJeu, 2))
input("------")
print("meilleur coup à jouer (4 coups à l'avance):", strategie(listeJeu, 3))
input("------")
```

Exercices “oral ESM” préparés en TP

Pelletier Sylvain PSI, LMSC

⋆ Étude de suites implicites

Exercice 1 Étude de suite implicite

Soit pour $n \geqslant 1$, la fonction :

$$f_n : x \longmapsto x + 1 - \frac{e^x}{n}$$

Le bue de cet exercice est d'étudier la suite implicite définie pour $n \geqslant 1$ par :

$$u_n < 0 \text{ et } f_n(u_n) = 0$$

1. **Étude graphique avec Python :**
   représenter graphiquement $f_1, \ldots, f_5$, trouver la bonne échelle pour déterminer graphiquement les valeurs de (u_n) pour $n = 1 \ldots 5$.
2. **Étude mathématique :** Montrer que pour tout $n \geqslant 1$, il existe un unique réel $u_n < 0$, tel que $f_n(u_n) = 0$.
3. Montrer que $u_n \in]-1, 0]$
4. **Étude numérique avec Python :** Sur Python : écrire une fonction permettant de calculer u_n à 10^{-3} près. Représenter quelques termes de la suite (u_n) ou simplement écrire leurs valeurs, conjecturer sa monotonie et sa convergence.
5. **Étude mathématique :** Démontrer la conjecture précédente.
6. **Étude mathématique :** Déterminer un équivalent de $u_n + 1$ puis un développement asymptotique de la suite (u_n).

Commentaire : relire la fiche sur les suites implicites.

Correction :

```
## import de pylab pour gagner du temps:
from pylab import *

# définition des contours du graphique
xmin = -1
xmax = 0
ymin = -1
ymax = 1

nbrPoints = 100

# def de la fonction:
def f(n,x):
    return x+1 - exp(x)/n

### dessin des premières fonctions
listeX = linspace(xmin, xmax , nbrPoints)

# copier / coller ou boucle for
listeY = [f(1,x) for x in listeX]
plot(listeX, listeY)
listeY = [f(2,x) for x in listeX]
plot(listeX, listeY)
listeY = [f(3,x) for x in listeX]
plot(listeX, listeY)
listeY = [f(4,x) for x in listeX]
plot(listeX, listeY)
listeY = [f(5,x) for x in listeX]
plot(listeX, listeY)

axis([xmin, xmax,ymin,  ymax])
```

```
grid() # pour afficher la grille

show()


def dico(n):
    ## Adapter l'algo, ne pas réciter
    a = -1
    b = 0
    while (b-a) > 10**(-4):
        c = (a+b)/2
        if f(n,c) >0 :
            b=c
        else:
            a=c
    return c


# affichage des valeurs:
for i in range(1, 10):
    print("u_", i, "=", dico(i))

# dessin:
for i in range(1, 10):
    plot(i, dico(i), "o")
show()
```

1.
2. C'est le théorème des valeurs intermédiaires. La dérivée $f_n' : x \longmapsto 1 - \frac{e^x}{n}$ est strictement positive sur $\mathbb{R}_-$. $\lim_{x\to-\infty} f_n(x) = -\infty$, donc le thm s'applique (continuité + stricte monotonie + tableau des variations).
   NB : dessiner le tableau des variations et le compléter au fur et à mesure
3.
4. $f_n(-1) = -\frac{e^{-1}}{n} < 0$. Ne pas oublier de mettre cette valeur dans le tableau des variations.
5. C'est la dichotomie.
6.
7. On calcule $f_n(u_{n+1})$:

$$\begin{aligned} f_n(u_{n+1}) =& u_{n+1} + 1 - \frac{e^{u_{n+1}}}{n} \\ =& e^{u_{n+1}} \left(\frac{1}{n+1} - \frac{1}{n} \right) \text{ car } u_{n+1} + 1 = \frac{e^{u_{n+1}}}{n+1} \\ <& 0 \end{aligned}$$

   D'où $f_n(u_{n+1}) < f_n(u_n)$ et f_n est strictement croissante. D'où : $u_{n+1} < u_n$. ainsi (u_n) est décroissante et minorée par -1. Elle converge vers une limite l, on passe ensuite à la limite :

$$u_n + 1 - \frac{e^{u_n}}{n} = 0$$

   Or :

$$\lim_{n\infty} \frac{e^{u_n}}{n} = 0$$

   Ainsi :

$$l + 1 = 0 \text{ ie } l = -1.$$

8. On a :

$$u_n + 1 = \frac{e^{u_n}}{n} \to \frac{e^{-1}}{n}$$

donc :

$$u_n = -1 + \frac{e^{-1}}{n}$$

Cela donne :

$$u_n = -1 + \frac{e^{-1}}{n} + \underset{n\infty}{o}\left(\frac{1}{n}\right)$$

Exercice 2 **Étude d'une suite implicite**

Soit $n \in \mathbb{N}^*$.

1. Montrer que l'équation $\tan x = x$ admet une unique solution dans l'intervalle

$$I_n = \left]-\frac{\pi}{2} + n\pi, \frac{\pi}{2} + n\pi\right[.$$

   On note cette solution x_n.
2. Proposer une résolution graphique pour déterminer les valeurs de x_n.
3. Sur Python : écrire une fonction qui calcule une approximation de x_n en fonction de n.
   Conjecturer alors $\lim\limits_{n\infty} x_n$, et $\lim\limits_{n\infty} x_n - n\pi$.
4. Montrer que $x_n \underset{n\to+\infty}{\sim} n\pi$,
5. Montrer que : $x_n - n\pi = \arctan(x_n)$ En déduire $\lim\limits_{n\infty} x_n - n\pi$.
6. Montrer que $x_n = \frac{\pi}{2} - \arctan\left(\frac{1}{x_n}\right) + n\pi$.
7. En déduire $\left(x_n - \frac{\pi}{2} - n\pi\right) \underset{n\to+\infty}{\sim} -\frac{1}{n\pi}$,

```
from pylab import *



nMax = 10
xmin = -pi/2
xmax = pi/2 + nMax*pi
ymin= -10
ymax = 10
nbrPoints = 100
eps = 10**(-10) # marge de sécurité pour le bords


for i in range(nMax):
    gauche = - pi/2 + i*pi
    droit = pi/2 + i*pi
    listeX = linspace(gauche + eps, droit -eps , nbrPoints)
    listeY = [ tan(x) for  x in listeX]
    plot(listeX, listeY)

# représentation y=x
# REM il y a plus rapide mais bon
listeX = linspace(xmin, xmax, nbrPoints )
listeY = linspace(xmin, xmax, nbrPoints )
plot(listeX, listeY)


axis([xmin, xmax, ymin, ymax])
grid()
show()

def f(x):
    return tan(x) -x
```

```
def dico(n):
    a = - pi/2 + i*pi + eps
    b = pi/2 + i*pi - eps
    while (b-a)> eps :
        c = (a+b)/2
        if f(c)>0 :
            b = c
        else :
            a = c
    return (a+b)/2

listeU = [0] # attention au décalage d'indice
for i in range(1,nMax):
    listeU.append(dico(i))

for i in range(1,nMax):
    print("i:",i," ", listeU[i], "ui -npi:", listeU[i] - i*pi)

listeY = [listeU[i] - i*pi for i in  range(1, nMax) ]

plot(range(1, nMax), listeY, "o")
grid()
show()
```

Commentaire : étude de suite implicite mais cette fois c'est l'intervalle qui dépend de n.

Correction :

1. C'est le théorème des valeurs intermédiaires : La fonction $f : x \longmapsto \tan x - x$ est strictement croissante sur chaque intervalle I_n, puisque sa dérivée est $x \longmapsto \tan^2(x)$. Ainsi, sur l'intervalle I_n, la fonction f est continue, strictement monotone et va de $-\infty$ à $+\infty$.
   Dessiner le tableau des variations sur l'intervalle I_n.
2. Attention, f n'existe pas au bords de I_n.
3.
4. C'est évident, car :

   $$\forall n \in \mathbb{N},\ x_n \in I_n$$

   et donc :

   $$-\frac{\pi}{2} + n\pi \leqslant x_n \leqslant \frac{\pi}{2} + n\pi$$

   ce qui donne l'équivalent. En particulier, $\lim\limits_{n\infty} x_n = \dfrac{\pi}{2}$
5. On a pour tout n :

   $$\tan x_n = x_n$$

   on aimerait appliquer l'arctangente, mais il faut être dans le bon intervalle :

   $$\tan\left(x_n - n\pi\right) = x_n$$

   Cette fois, on a bien $x_n - n\pi$ entre $-\frac{\pi}{2}$ et $\frac{\pi}{2}$, donc :

   $$x_n - n\pi = \arctan(x_n)$$

   Comme x_n tend vers $+\infty$, on a :

   $$\lim_{n\infty} x_n - n\pi = \frac{\pi}{2}.$$

6. On utilise ensuite :

   $$\forall x > 0,\ \arctan(x) = \frac{\pi}{2} - \arctan\left(\frac{1}{x}\right)$$

Cela donne :

$$x_n = n\pi + \frac{\pi}{2} - \arctan\left(\frac{1}{x_n}\right)$$

Comme $\arctan u \underset{u\to 0}{\sim} u$, cela donne :

$$x_n - n\pi - \frac{\pi}{2} = -\arctan\left(\frac{1}{x_n}\right) \underset{n\to\infty}{\sim} -\frac{1}{x_n} \underset{n\to\infty}{\sim} -\frac{1}{n\pi}$$

✯ Dénombrements

Exercice 3 Soit n un entier non nul. On désigne par u_n le nombre de listes de n termes, chaque terme étant 0 ou 1, et n'ayant pas deux termes 1 consécutifs.

1. Que vaut u_1 ? u_2 ?
2. Démontrer que pour tout $n \geqslant 3$, on a $u_n = u_{n-1} + u_{n-2}$.
3. Écrire un algorithme qui permet de calculer u_{20}.
4. **Application :** un concours comporte vingt questions, numérotées de 1 à 20. On a constaté que, parmi les 17712 personnes ayant participé au concours, aucune n'a répondu juste à deux questions consécutives.
   Peut-on affirmer que deux candidats au moins ont répondu de la même manière au questionnaire, c'est-à-dire juste aux mêmes questions et faux aux mêmes questions ?

Exercice 4 Dans ma maison, il y a un escalier de 17 marches. Pour descendre cet escalier, je peux à chaque pas descendre une marche, descendre deux marches, ou descendre trois marches à la fois. Combien y-a-t-il de façons de descendre cet escalier ?

Indication : notez $S(n)$ le nombre de façons de descendre un escalier de taille n et trouver une formule de récurrence.

✯ Simulation numérique d'expérience aléatoire

Exercice 5 On dispose de deux boîtes d'allumettes : la boîte « pile » et la boîte « face », et d'une pièce qui donne « pile » avec une probabilité p.

Au départ, les deux boîtes contiennent n allumettes.
On tire à pile ou face, si on obtient pile, on enlève une allumette de la boîte « pile », idem si on obtient face.
On s'arrête lorsqu'une boite est vide, et on note X le nombre d'allumettes dans la boîte non vide.
Ainsi, X est une valeur aléatoire de $[\![1,n]\!]$.

1. Écrire une fonction `simule(n, p)` qui calcule la valeur de X (aléatoire donc).
   On utilisera la fonction `rand`.
   REM : on testera avec $n = 8$ et $p = \frac{1}{3}$.
2. Écrire une fonction `simuleMFois(n,p, m)` qui réalise m expériences et qui sort une liste L, tel que `L[i]` est l'estimation empirique de $p(X = i)$, ie le nombre de simulations pour lesquelles le résultat a été i divisé par m (le nombre total de simulations).
3. En utilisant la fonction `bar` du module `pylab`, dessiner un diagramme en bâton des valeurs empiriques de $p(X = i)$.
4. Calculer mathématiquement la valeur exacte de $p(X = i)$.
   Indication : si $(X = k)$, c'est que l'on a fait $2n - k$ expérience. Si la dernière a fini sur pile, alors on a eu $n - 1$ pile dans les tirages 1 à $2n - k - 1$ pour finir sur pile.
5. Écrire une fonction `loiExacte(n,p)` qui sort une liste L tel que `L[i]` est la valeur exacte de $p(X = i)$.
6. Comparer la loi exacte et la loi approchée.

Correction :

```
# Modules et fonctions importés
################################
from pylab import *

# Fonctions
###########

```

```
def simule(n,p):
    """
    entrée: n = int
              = taille des boîtes.
            p = float de ]0,1[
              = proba d'avoir pile
    sortie: X = entier
              = nbr d'allumettes dans la boîte non vide lorsque l'une est vide
    """
    nbrPile = n
    nbrFace = n

    while nbrFace >0 and nbrPile >0 :
        alea =rand()
        if alea < p : #pile
            nbrPile -= 1
        else :
            nbrFace -= 1
    if (nbrFace>0):
        return(nbrFace)
    else:
        return(nbrPile)

def simuleMFois(m, n, p):
    """
    entrée: n = int
              = taille des boîtes.
            p = float de ]0,1[
              = proba d'avoir pile
            m =  entier = nbr d'expériences
    sortie: L = liste de n float
              = L[i] est la proba mesurée telles que X=i
    """
    L = [0]*(n+1) # attention au +1
    for i in range(m) :
        X = simule(n,p)
        L[X] += 1
    L = [x/m for x in L]
    return(L)

def combi(k,n):
    """
    entrée: k = int
            n = int
    sortie int = valeur de k parmis n
    """
    # num = n! / (n-k)!
    num = 1
    for i in range(n, n-k, -1):
        num *= i
    # denom = k!
    denom = 1
    for i in range(1,k+1):
        denom *= i
    return num/denom




def loiExacte(n, p):
    """
    entrée: n = int
              = taille des boîtes.
```

```
            p = float de ]0,1[
74            = proba d'avoir pile
    sortie: L = liste de n float
76            = L[i] est la proba exacte de  X=i
    """
78  q= 1-p
    L = [0] *(n+1)
80  for k in range(1, n+1):
        L[k] = combi(n-1,2*n-k-1) * p**(n-k) * q**(n-k) * (p**k + q**k )
82  return L


84



86
# Code:
88 #######

90 m = 1000000
n = 8
92 p = 1/3

94 L = simuleMFois(m,n,p)
Le = loiExacte(n,p)
96 print("\n"+"-"*10+"résultats obtenus"+"-"*10)
for i in range(n+1):
98     print("i=", i," proba mesurée=", L[i], " proba exacte=", Le[i])

100 bar(range(len(L)), L)
plot(range(len(L)), Le, color = "red")
102 show()
```

Pour les mathématiques, on a :

$$p(X=k) = p(X=k \cap P_k) + p(X=k \cap F_k)$$

On calcule le premier terme. Pour cela, on note Y_j le nombre de pile dans les tirages 1 à j, on a :

$$(X=k \cap P_k) = (Y_{2n-k-1} = n-1 \cap P_k)$$

or $Y_{2n-k-1} = n-1$ et P_k sont indépendants (puisque Y_{2n-k-1} n'est fonction que des tirages 1 à $2n-k-1$). Ainsi :

$$\begin{aligned} p(X=k \cap P_k) &= p(Y_{2n-k-1} = n-1)p(P_k) \\ &= \binom{2n-k-1}{n-1} p^{n-1}q^{n-k}p \\ &= \binom{2n-k-1}{n-1} p^n q^{n-k} \end{aligned}$$

De même :

$$p(X=k \cap P_k) = \binom{2n-k-1}{n-1} q^n p^{n-k}$$

Ainsi :

$$\begin{aligned} p(X=k) &= \binom{2n-k-1}{n-1} \left(p^n q^{n-k} + q^n p^{n-k}\right) \\ &= \binom{2n-k-1}{n-1} p^{n-k}q^{n-k} \left(p^k + q^k\right) \end{aligned}$$

Exercice 6 On considère une urne contenant n boules numérotées de 1 à n.

On extrait successivement une à une toutes les boules de l'urne.

On note, pour $i \in [\![1,n]\!]$,

- Z_i la variable aléatoire égale au numéro obtenu lors du tirage i.
- $X_i = \max(Z_1, Z_2, \ldots, Z_i)$,
- B_i la variable aléatoire égale à 1 si le tirage i est un « *record* », c'est-à-dire que le numéro de la boule tiré au tirage i est strictement supérieur à tous les précédents, *i.e.* si $Z_i > X_{i-1}$. On pose que B_1 est la variable certaine égale à 1.
- Enfin, on désigne par R la variable aléatoire : $R_n = \sum_{i=1}^{n} B_i$, ie le nombre de record.

1. Écrire une fonction `tirage(n)` qui prend en entrée n et qui sort un tirage des boules 1 à n successivement et sans remise sous la forme d'une liste Z.
   Ainsi, $Z[i]$ est une simulation de la variable Z_i.
   Tester avec $n = 10$.
   Indication : attention au décalage, car il n'y a pas de tirage 0. On peut prendre comme convention que $Z[0]$ est nul.
2. Écrire une fonction `trouveRecord(Z)`, qui renvoie un vecteur B, avec $B[i] = 1$ si dans le tirage Z le i-ième nombre tiré est un record, 0 sinon.
   Indication : on pourra prendre la convention que $B[0]$ est nul, mais $B[1]$ doit être égal à 1.
3. Écrire une fonction `estimeLoiX(n,m)`, qui réalise m simulations et qui renvoie une liste L avec $L[i]$ est le nombre de simulations où la variable B_i a été égale à 1 divisé par le nombre total de simulations.
4. Déterminer mathématiquement $E(B_i)$ et comparer.
5. Écrire une fonction `estimeNbrRecord(n,m)`, qui fait m expérience et calcule la valeur moyenne de R_n sur ces m expériences.
6. Calculer $E(R_n)$ mathématiquement et comparer.

Correction :

```
from pylab import *


def tirage(n):
    """
    entrée: n = int = nbr de boules
    sortie: Z = list de int de longueur n+1
              = liste aléatoire des boules tirées sans remise
              = Z[i] est la simulation de Zi
    NB: par convention Z[0] = 0 (pas de premier tirage)
    """

    urne = list(range(1,n+1))
    Z = [0]
    for i in range(n):
        alea = int(rand() * len(urne))
        Z.append(urne.pop(alea))
    return Z

def trouveRecord(Z):
    """
    entrée: Z = list de int de longueur n+1
              = liste aléatoire des boules tirées sans remise
              = Z[i] est la simulation de Zi
    sortie: B = liste de 0/1
              = B[i] = 1 si le i-ième est un record
    """

    B = [0]*len(Z)
    B[0] = 0
    max = Z[0]
    for  i in range(1, len(Z) ):
        if Z[i] > max :
            max = Z[i]
```

```
            B[i] = 1
    return B

def estimeLoiX(n,m):
    """
    entrée: n = int = nbr de boules
            m = int = nbr de simulation
    sortie: list de float donnant la proba empirique de Xi= 1
    """
    loi = [0]* (n+1)
    for exp  in range(m):
        B = trouveRecord(tirage(n))
        for i in range(n+1):
            loi[i] += B[i]
    return [eff/ m for eff in loi]

def loiExacte(n):
    """
    entrée: n = int = nbr de boules
    sortie: liste de float, vrai proba que xi =1
    NB: on ajoute 0 pour le tirage 0
    """
    loi = [0] + [ 1/i for i in range(1,n+1)]
    return loi


def estimeNbrRecord(n,m) :
    """
    entrée: n = int = nbr de boules
            m = int = nbr de simulation
    sortie: float: nbr de records moyen sur m expérience
    """
    nbrRecord = 0
    for exp  in range(m):
        B = trouveRecord(tirage(n))
        for i in range(n+1):
            nbrRecord += B[i]
    return nbrRecord /m




print('--- exercices sur les records tirage sans remise ---')

n = 10
print('--- test sur un tirage ---')
Z = tirage(10)
print("le tirage aléatoire:",Z)
B = trouveRecord(Z)
print("les records:", B)

print('--- simulation de plusieurs tirages ---')
m = 10000
loiE = estimeLoiX(n,m)
bar(range(len(loiE)), loiE)
loiV = loiExacte(n)
plot(range(len(loiV)), loiV, color= "red")
title("loi empirique / calculée des records pour chaque i")
show()

figure()

print('--- recherche du nombre de record en fonction de n ---')
```

```
nMax = 30
nMin = 3
print('--pour n entre ', nMin, ' et ', nMax, '---')
nbrR = [ estimeNbrRecord(n,m) for n in range(nMin, nMax)]
vraiR = [ sum ( [1/i for i in range(1, n+1)])  for n in range(nMin, nMax)]
plot(range(nMin, nMax), nbrR, color = 'blue')
plot(range(nMin, nMax), vraiR, color = 'red')
title("nbr de records empirique / calculé en fonction de n ")
show()
```

On calcule $p(B_i = 1)$, c'est la probabilité d'avoir un record au numéro i.

Méthode 1 : On peut procéder par dénombrements. On ne s'intéresse qu'au i premier tirage, ainsi, l'univers est l'ensemble des listes sans répétitions de longueur i. On a donc :

$$card(\Omega) = \frac{n!}{(n-i)!}$$

De plus, un tirage où $B_i = 1$ est déterminé par :

- le choix des i boules que l'on va placer : $\binom{n}{i}$ choix.
- La plus grande est placée en dernier (position i), et il reste à placer les autres. Pour cela, on doit donc placer $i-1$ boules dans $i-1$ places. Ainsi, il y a $(i-1)!$ choix.

On obtient :

$$\begin{aligned} p(B_i = 1) &= \frac{\binom{n}{i}(i-1)!}{\frac{n!}{(n-i)!}} \\ &= \frac{n!(n-i)!(i-1)!}{n!i!(n-i)!} = \frac{1}{i} \end{aligned}$$

Méthode 2 : SCE associé à Z_i la valeur de la i ième boule . On a :

$$\begin{aligned} p(B_i = 1) &= \sum_{k=1}^{n} p(B_i = 1 \cap Z_i = k) \\ &= \sum_{k=i}^{n} p(B_i = 1 \cap Z_i = k) \qquad \text{termes nuls} \end{aligned}$$

puis :

$$\begin{aligned} p(B_i = 1 \cap Z_i = k) &= p(Z_1 < k \cap Z_2 < k \cap \cdots \cap Z_{i-1} < k \cap Z_i = k) \\ &= \frac{k-1}{n}\frac{k-2}{n-1}\cdots\frac{k-(i-2)}{n-(i-2)}\frac{1}{n-(i-1)} \\ &= \frac{k-1}{n}\frac{k-2}{n-1}\cdots\frac{k-i+2)}{n-i+2)}\frac{1}{n-i+1)} \\ &= \frac{(k-1)!}{(k-i+1)!}\frac{(n-i)!}{n!} \end{aligned}$$

On obtient :

$$\begin{aligned} p(B_i = 1) &= \sum_{k=i}^{n} \frac{(k-1)!}{(k-i+1)!}\frac{(n-i)!}{n!} \\ &= \frac{1}{\binom{n}{i}}\sum_{k=i}^{n}\frac{(k-1)!}{i!(k-i+1)!} \\ &= \frac{1}{\binom{n}{i}}\frac{1}{i}\sum_{k=i}^{n}\frac{(k-1)!}{(i-1)!(k-i+1)!} \\ &= \frac{1}{\binom{n}{i}}\frac{1}{i}\sum_{k=i}^{n}\binom{k-1}{i-1} \end{aligned}$$

Il faut ensuite vérifier (exercice classique par récurrence) que :

$$\sum_{k=i}^{n}\binom{k-1}{i-1} = \frac{1}{\binom{n}{i}}$$

On trouve donc :

$$B_i \sim \mathscr{B}(\frac{1}{i})$$

et

$$E(R_n) = \sum_{i=1}^{n} \frac{1}{i}$$

⋆ Étude de suites récurrentes et calculs de somme

Exercice 7 Considérons la suite $(a_n)_{n\in N}$ définie par $a_0 = 1$ et la relation :

$$\forall n \in \mathbb{N},\ a_{n+1} = 1 - e^{-a_n}$$

1. **En python :** Écrire une fonction récursive `a(n)` qui renvoie la valeur de a_n.
2. **En python :** Écrire une fonction `listeA(n)` de complexité $O(n)$ qui renvoie une liste des n premières termes de la suite (a_n).
3. **En python :** Représenter (a_n) en fonction de n. Que peut-on conjecturer sur la convergence de la suite (a_n) ?
4. Le démontrer.
5. Pour tout $n \in \mathbb{N}$, on note $S_n = \sum_{k=0}^{n} a_k^2$.
   En python : Écrire une fonction `listeS(n)` de complexité $O(n)$ qui renvoie une liste des n premiers termes de la suites (S_n).
6. **En python :** Représenter (S_n) en fonction de n. Que peut-on conjecturer sur la convergence de la suite (S_n) ?
7. Le démontrer.
   Indication : trouver un équivalent de $a_{n+1} - a_n$.

Correction :

```
from pylab import *

def a(n):
    """
    entrée: n = int
    sortie: float =    valeur de an
    fonction récursive
    """
    if n==0 :
        return 1
    else :
        return 1 - exp(-a(n-1))


def listeA(n):
    """
    entrée: n = int
    sortie: L = list de float
    = list des valeurs de an
    """
    L = [0]*n
    L[0] = 1
    for i in range(n-1):
        L[i+1] = 1 - exp(-L[i])
    return L

def listeS(n):
    """
    entrée: n = int
    sortie: S = list de float
    = list des valeurs de Sn
    """
    L = listeA(n)
```

```
    S = [0]*n
    S[0] = L[0]
    for i in range(1,n):
        S[i] = S[i-1] + L[i]**2
    return S

print("test de la fonction récursive")
for i in range(5):
    print("i=",i, "ai=", a(i))
N = 150

print("test de la fonction listeA")
listeY = listeA(N)
for n in range(N):
    print("n=",n, "an=", listeY[n])

print("représentation graphique des termes (an)")
listeX = list(range(N))
plot(listeX, listeY, 'o')
show()

print("test de la fonction listeS")
listeY = listeS(N)
for n in range(N):
    print("n=",n, "Sn=", listeY[n])
print("représentation graphique des termes (Sn)")
listeX = list(range(N))
plot(listeX, listeY, 'o')
show()
```

On montre tout d'abords par récurrence la propriété :

$$\mathscr{P}(n) : "a_n \in [0,1]"$$

Pour $n = 0$, on a bien $\mathscr{P}(0)$.

Soit n fixé, tel que $\mathscr{P}(n)$ est vraie. On a alors :

$$0 \leqslant a_n \leqslant 1$$
$$0 \leqslant 1 - e^{-a_n} \leqslant 1 - e^{-1}$$

puisque $x \mapsto 1 - e^{-x}$ est croissant. Ainsi : $a_{n+1} \in [0,1]$ et l'hérédité puis la conclusion.

Une propriété classique de convexité indique $e^x \geqslant 1 + x$, avec égalité si et seulement si $x = 0$ Ainsi :

$$e^{-a_n} \geqslant 1 - a_n$$

et donc : $a_n \geqslant 1 - e^{-a_n}$ ie $a_n \geqslant a_{n+1}$ et la suite (a_n) est décroissante.

Décroissante et minorée par 0, elle converge donc vers une limite $l \in [0,1]$.

On passe à la limite dans l'égalité :

$$\forall n \in N,\ a_{n+1} = 1 - e^{-a_n}$$

Cela donne :

$$l = 1 - e^{-l}$$

Ainsi : $e^{-l} = 1 - l$ et donc $l = 0$.

La suite a_n converge donc vers 0.

$$e^{-u} = 1 - u + \frac{u^2}{2} + \underset{u \to 0}{o}\left(u^2\right)$$

On a ensuite :

$$\lim_{n\infty} a_n = 0 \text{ et } 1 - e^{-u} - u \underset{u \to 0}{\sim} -\frac{u^2}{2}$$

Ainsi :

$$a_{n+1} - a_n = 1 - e^{-a_n} - a_n \underset{n\infty}{\sim} -\frac{a_n^2}{2}$$

Ainsi, par critère de comparaison des séries à termes de signe constant (ici négatif), on a que la nature de la série $\sum a_n^2$ est la même que la série $\sum a_{n+1} - a_n$. Cette dernière est une série télescopique donc converge puisque (a_n) tend vers une limite finie. Ainsi, la série (S_n) converge.

Vers quoi ?

✯ Étude de suites d'intégrales et calculs de somme

Exercice 8 Pour $n \in \mathbb{N}$, on pose $u_n = \int_0^1 x^n e^{-x} dx$

1. **Avec Python :** calculer u_n.
   Représenter les premiers termes de la suite et conjecturer $\lim_{n\infty} u_n$.
2. Justifier l'existence et calculer $\lim_{n\infty} u_n$.
   REM : peut se faire avec le thm de convergence dominée. Sinon, passer directement à la question suivante.
3. Donner un encadrement simple de u_n en fonction de n.
4. En déduire la nature des séries $\sum u_n$ et $\sum \frac{u_n}{n}$.
5. **Avec Python :** représenter les premiers valeurs de $v_n = \sum_{k=0}^{n} \frac{u_k}{k!}$.
   Conjecture sur la convergence ?
6. Le démontrer.
   REM : utilise le thm de convergence dominée.

```python
from pylab import *

def u(n):
    nbrPoints = 100
    x = linspace(0,1, nbrPoints)
    pas = x[1] - x[0]
    u = 0
    for i in range(nbrPoints-1):
        u += x[i]**n * exp(-x[i])
    u = u *pas
    return u


def v(n) :
    # initialisation:
    listeV = [0]*n
    facto = 1
    listeV[0] = u(0)
    for k in  range(1,n):
        facto = facto *k
        listeV[k] = listeV[k-1] + u(k)/facto
    return listeV

nbrTermes = 10
listeU = [u(n) for n  in range(nbrTermes)]
for i in range(nbrTermes):
    print("i=", i, "ui=", listeU[i])

plot( list(range(nbrTermes)), listeU, 'o')
show()

listeV = v(nbrTermes)
for i in range(nbrTermes):
    print("i=", i, "vi=", listeV[i])

```

```
plot( list(range(nbrTermes)), listeV, 'o')
show()
```

Autre idée (extrait) :

```
def trapeze(f,a,b,N):
    """
    entrée: f = fct R -> R
            (a,b) = float
            N = int
    sortie: float = valeur de intégrale de a à b de f
            calcul par trapeze
    """
    t = linspace(a,b, N)
    delta = (b-a)/(N-1)
    I = ( f(t[0]) + f(t[N-1]) ) /2
    for i in range(1, N-1):
        I += f(t[i])
    return delta*I


def calculeI(n):
    """
    entrée: n = int
    sortie: list de float = valeur de I(n)
            calculé par les trapèzes
    """
    N = 1000


    I = [0 for i in range(n)]
    for i in range(n):
        def f(x):
            return x**i *exp(-x)
        I[i] = trapeze(f, 0, 1, N)

    return I
```

Correction :

1. Il faut utiliser la méthode des rectangles ou des trapèzes.
2. Avec le thm de convergence dominé. On note :

$$f_n : x \longmapsto x^n e^{-x}$$

on a :

$$\forall n \in \mathbb{N},\ |f_n(x)| \leqslant 1 \text{ intégrable sur }]0,1[$$
$$\forall x \in]0,1[, \lim_{n\infty} f_n(x) = 0$$

D'où le résultat.

3. On a :

$$\forall x \in [0,1],\ x^n \leqslant x^n e^{-x} \leqslant x^n e^{-1}$$

donc en intégrant :

$$\frac{1}{n+1} \leqslant u_n \leqslant \frac{e^{-1}}{n+1}$$

on obtient $\lim_{n\infty} u_n = 0$.

4. D'où $\sum u_n$ diverge et $\sum \frac{u_n}{n}$ converge.
5.

6. Le plus simple c'est avec le thm de convergence dominé sur la somme partielle :

$$g_n : x \longmapsto \sum_{k=0}^{n} \frac{x^k}{k!} e^{-x}$$

On a alors :

$$\begin{aligned} \forall x \in]0,1[,\ |g_n(x)| &= \sum_{k=0}^{n} \frac{x^k}{k!} e^{-x} \\ &\leqslant \sum_{k=0}^{+\infty} \frac{x^k}{k!} e^{-x} = 1 \end{aligned}$$

et à x fixé, on a $\lim_{n\infty} g_n(x) = 1$.
Ainsi, on peut appliquer le thm de convergence dominée :

$$\begin{aligned} \lim_{n\infty} v_n &= \lim_{n\infty} \int_0^1 \sum_{k=0}^{n} \frac{x^k}{k!} e^{-x} dx \\ &= \lim_{n\infty} \int_0^1 g_n(x) dx \\ &= \int_0^1 \lim_{n\infty} g_n(x) dx = 1. \end{aligned}$$

✯ Étude de suites récursives

Exercice 9 Soit la fonction :

$$f : \left\{ \begin{array}{ccc} [0,1] & \to & \mathbb{R} \\ x & \longmapsto & \frac{3}{2}x - x^3 \end{array} \right.$$

On considère la suite (u_n) définie par $u_0 \in [0,1]$ et la relation :

$$\forall n \in \mathbb{N},\ u_{n+1} = \frac{3}{2} u_n - u_n^3$$

1. **En python :** conjecturer le comportement de (u_n) en fonction de $u_0 \in [0,1]$.
   Représenter le graphe de la fonction f et de la droite $y = x$ sur une même figure.
   Représenter $n \mapsto u_n$ pour différentes valeurs de u_0.
2. Pour $u_0 \in \left]0, \frac{1}{\sqrt{2}}\right[$, démontrer la conjecture précédente.
3. Factoriser $X^3 - \frac{3}{2}X + \frac{1}{\sqrt{2}}$
4. Montrer que

$$\frac{1}{\sqrt{2}} - u_{n+1} \underset{n\infty}{\sim} \frac{3}{\sqrt{2}} \left(u_n - \frac{1}{\sqrt{2}} \right)^2$$

```python
from pylab import *

def listeU(n, u0):
    """
    entrée: n = int = nbr de termes à calculer
            u0 = float = valeur initiale
    sortie: L = liste des valeurs de u(n)
    """
    L = [0]*n
    L[0] = u0
    for i in range(1, n):
        L[i] = (3/2)*L[i-1] - L[i-1]**3
    return L


nbrTermes = 30
```

```
u0 = 0.001
L = listeU(nbrTermes, u0)

for i in range(nbrTermes):
    print("i=", i, "ui=", L[i])

# exemple de dessin
plot(L,L, 'o')
listeX = linspace(0,1, 100)
listeY = [ 3/2 *x - x**3 for x in listeX]
plot(listeX, listeY)
plot(listeX, listeX)
show()

# Dessin plus évolué
listeX = []
listeY = []
for i in range(nbrTermes -1):
    listeX.append(L[i])
    listeY.append(L[i])
    listeX.append(L[i])
    listeY.append(L[i+1])
    listeX.append(L[i+1])
    listeY.append(L[i+1])

plot(listeX,listeY)
listeX = linspace(0,1, 100)
listeY = [ 3/2 *x - x**3 for x in listeX]
plot(listeX, listeY)
plot(listeX, listeX)
show()
```

Correction :

1.
2. On démontre d'abord que l'intervalle $\left]0, \frac{1}{\sqrt{2}}\right[$ est stable.

   Pour cela, il suffit d'étudier la fonction f et de constater que f est croissante sur $\left]0, \frac{1}{\sqrt{2}}\right[$ (en dérivant) et que $f(0) = 0$ et que $f\left(\frac{1}{\sqrt{2}}\right) = \frac{1}{\sqrt{2}}$. On dessine le tableau des variations.

   Il est donc clair (par récurrence immédiate), que $u_n \in \left]0, \frac{1}{\sqrt{2}}\right[$ pour toute valeur de n.

   Ensuite (et seulement ensuite), on regarde $u_{n+1} - u_n$:

$$\begin{aligned} u_{n+1} - u_n &= \frac{1}{2}u_n - u_n^3 \\ &= u_n\left(\frac{1}{2} - u_n^2\right) \end{aligned}$$

   avec ce que l'on a vu, il devient évident que $u_{n+1} - u_n \geqslant 0$.

   donc (u_n) est croissante et majorée (par $\frac{1}{\sqrt{2}}$, encore l'intervalle stable). Donc (u_n) converge vers l, avec

$$l \in \left[0, \frac{1}{\sqrt{2}}\right]$$

   (toujours l'intervalle stable).

   On a :

$$l = \frac{3}{2}l - l^3$$

   il faut donc vérifier que $l \neq 0$ (ce qui se fait en vérifiant que $u_0 > 0$). puis, on a :

$$0 = \frac{1}{2} - l^2$$

   et donc puisque $l \in \left[0, \frac{1}{\sqrt{2}}\right]$ on en déduit $l = \frac{1}{\sqrt{2}}$.

3. On constate que $\frac{1}{\sqrt{2}}$ est racine du polynôme $P = X^3 - \frac{3}{2}X + \frac{1}{\sqrt{2}}$. On dérive et on a :

$$P' = 3\left(X^2 - \frac{1}{2}\right)$$

Ainsi, $\frac{1}{\sqrt{2}}$ est racine double. Donc on peut écrire :

$$X^3 - \frac{3}{2}X + \frac{1}{\sqrt{2}} = \left(X - \frac{1}{\sqrt{2}}\right)^2 (X - \alpha)$$

on veut que $\alpha + 2\frac{1}{\sqrt{2}} = 0$ et donc $\alpha = -\sqrt{2}$
On obtient :

$$X^3 - \frac{3}{2}X + \frac{1}{\sqrt{2}} = \left(X - \frac{1}{\sqrt{2}}\right)^2 \left(X + \sqrt{2}\right)$$

4. On écrit pour $n \in \mathbb{N}$:

$$\begin{aligned}\frac{1}{\sqrt{2}} - u_{n+1} &= \frac{1}{\sqrt{2}} - \frac{3}{2}u_n + u_n^3 \\ &= P(u_n) \\ &= \left(u_n - \frac{1}{\sqrt{2}}\right)^2 \left(u_n + \sqrt{2}\right) \\ &\underset{n\infty}{\sim} \left(\frac{1}{\sqrt{2}} + \sqrt{2}\right)\left(u_n - \frac{1}{\sqrt{2}}\right)^2 = \frac{3}{\sqrt{2}}\left(u_n - \frac{1}{\sqrt{2}}\right)^2\end{aligned}$$

✯ Travail sur les polynômes

Exercice 10

En Python, les polynômes seront représentés sous forme de liste.

Le polynôme $P = \sum_{k=0}^{n} a_k X^k$ sera ainsi représenté sous la forme de la liste de longueur $n+1$: `[a0, ..., an]`.

R Attention au décalage : si le polynôme est de degré n, la liste est de longueur $n+1$.

1. Écrire une fonction **évalue** qui prend en entrée un polynôme P et un réel x et qui calcule $P(x)$.
   On utiliser la méthode de Hörner :

   $$P(x) = ((\ldots((a_n x + a_{n-1})x + a_{n-2})\ldots)x + a_1)x + a_0$$

   Pour tester, afficher le graphe d'un polynôme.
2. Écrire une fonction Python **somme** qui prend en entrée deux polynômes P et Q et qui calcule : $P+Q$. (attention, les polynômes n'ont pas nécessairement le même degré).
3. Écrire une fonction Python **produit** qui prend en entrée deux polynômes P et Q et qui calcule : PQ.
4. Écrire une fonction Python **dérivation** qui prend en entrée un polynôme P et qui calcule : P'.
5. Écrire une fonction Python **décalage** qui prend en entrée un polynôme P, et qui calcule le polynôme $P(X+1)$.
   Indication : utiliser une formule de Taylor.
6. Soit $n \in \mathbb{N}$.
   Avec Python, donner la matrice de l'application :

   $$u : P \longmapsto (1 - X^2)P' + nXP$$

   dans la base canonique de $\mathbb{R}_n[X]$.

On pourra utiliser le code suivant (après l'avoir compris et éventuellement adapté) :

```
def affiche(P):
    """
    entrée: P = list
              = liste des coefficients
```

```
    sortie: rien affichage à l'écran
    """
    if len(P) == 0 :
        print("poly nul")
        return
    msg = str(P[0])
    for i in range(1, len(P)):
        if P[i] != 0 :
            msg = str(P[i])+"X**"+str(i)+" + "+ msg
    print(msg)
```

⋆ Pivot de Gauss

Exercice 11 Écrire une fonction qui prend en entrée une matrice A et qui calcule son inverse A^{-1} en utilisant la méthode de Gauss Jordan.

On utilisera diverses techniques pour le changement de ligne :

- prendre le premier pivot non nul.
  Dans ce cas, on vérifiera que sur une matrice aléatoire le premier pivot est toujours non nul.
- prendre le pivot le plus grand en valeur absolue.

⋆ Réduction des matrices

R La fonction `eig` qui calcule les valeurs propres et les vecteurs propres est à connaître !

Exercice 12

Soit $A \in \mathscr{M}_n(\mathbb{K})$ une matrice carrée. On suppose que A est diagonalisable et on note $Sp(A) = \{\lambda_1, \dots, \lambda_n\}$. On suppose aussi que la valeur propre λ_n de plus grand module est simple, ie $\forall i \in [\![1, n-1]\!]$, $|\lambda_i| < |\lambda_n|$.

Justifier alors que :

$$\lim_{p \to +\infty} \frac{Tr(A^{p+1})}{Tr(A^p)} = \lambda_n.$$

Application en informatique : en utilisant la fonction `dot(A,B)` qui calcule le produit de deux `2d-array` A et B, écrire une fonction donnant une approximation de λ_n.

Tester sur une matrice aléatoire.

⋆ Intégrales généralisées

Exercice 13 Pour $(t,u) \in (]0,+\infty[)^2$, calculer $\displaystyle\int_t^u \frac{\ln(x)}{x^a} dx$.

En déduire les résultats suivants :

$\displaystyle\int_0^1 \frac{\ln(x)}{x^a} dx$ converge si $a < 1$ et vaut $-\frac{1}{(1-a)^2}$

$\displaystyle\int_1^{+\infty} \frac{\ln(x)}{x^a} dx$ converge si $a > 1$ et vaut $\frac{1}{(1-a)^2}$

Avec Python, écrire une fonction qui prend en entrée a et qui calcule une approximation de $\displaystyle\int_0^1 \frac{\ln(x)}{x^a} dx$ proposer une méthode graphique pour vérifier le résultat obtenu.

Même question pour $\displaystyle\int_1^{+\infty} \frac{\ln(x)}{x^a} dx$

⋆ Méthode d'Euler pour la résolution d'équations différentielles

Principe de la méthode d'Euler explicite

On veut résoudre :

$$\begin{aligned} y' &= f(y,t) \text{ pour } t \in [a,b] \\ y(a) &= y_0 \text{ donné} \end{aligned}$$

La méthode est :

- On divise l'intervalle $[a,b]$ en $N+1$ points équirépartis :

$$t_0 = a < t_1 < \cdots < t_N = b \text{ on note } \Delta = t_{i+1} - t_i$$

- On calcule un vecteur Y de taille $N+1$, avec

$$Y[i] \approx y(t_i) \text{ pour } i \in [\![0,N]\!]$$

  Pour cela, on procède avec une boucle `for` :
  - on connaît la valeur de $Y[0] = y_0$,
  - on remplace avec l'approximation d'Euler :

$$y'(t_i) \approx \frac{y(t_{i+1}) - y(t_i)}{\Delta} \approx \frac{Y[i+1] - Y[i]}{\Delta}$$

- dans le cas d'une équation d'ordre 2, on a plusieurs choix :
  - utiliser un vecteur de fonction inconnues : $X(t) = \begin{pmatrix} y(t) \\ y'(t) \end{pmatrix}$
  - ou on remplace avec l'approximation d'Euler :

$$y''(t_i) \approx \frac{y(t_{i+1}) - 2y(t_i) + y(t_{i-1})}{\Delta^2}$$

Exercice 14 Résoudre numériquement le problème de Cauchy :

$$\begin{aligned} y' + 2y &= 2x^2 + (2x+1+\cos 3x)e^{-2x} \\ y(0) &= 0 \end{aligned}$$

sur un intervalle bien choisi.

Comparer à la solution exacte :

$$y : x \longmapsto x^2 - x + \frac{1}{2} + x(x+1)e^{-2x} + \frac{1}{3}\sin(3x)e^{-2x} - \frac{1}{2}e^{-2x}$$

Exercice 15 Résoudre numériquement le problème de Cauchy suivant :

$$\begin{aligned} y'' + y &= \cos x + \sin 2x \\ y(0) &= 1 \\ y'(0) &= 0 \end{aligned}$$

Comparer à la solution exacte :

$$x \mapsto -\frac{\sin 2x}{3} + \cos(x) + \left(\frac{2}{3} + \frac{x}{2}\right)\sin(x)$$
