Table des matières

1 **Révisions** . . . 3

I Théorie des algorithmes **3**

I.1 Notion d'algorithme . . . 3

I.2 Preuves formelles sur les algorithmes . . . 4

I.3 Exemples d'étude théorique d'algorithme . . . 5

II Fonctionnement de Python **9**

II.1 Architecture des ordinateurs . . . 9

II.2 Architecture logicielle . . . 10

II.3 Variables et objets en Python . . . 10

II.4 Représentation des nombres . . . 20

III Programmer en Python **30**

III.1 Variables en Python . . . 32

III.2 Les structures de contrôles en Python . . . 33

III.3 Conteneurs : str, list et array . . . 36

III.4 Les fonctions . . . 43

IV Algorithmes à connaître **46**

V Ingénieurie numérique **50**

V.1 Calcul de moyenne et de variance . . . 50

V.2 Calcul de π par la méthode de Monte-Carlo . . . 51

V.3 Calcul approché d'intégrale . . . 52

V.4 Résolution de $f(x) = 0$ . . . 52

V.5 Recherche . . . 55

V.6 Équation linéaire . . . 56

V.7 Autour des polynômes . . . 58

V.8 Méthode d'Euler . . . 59

V.9 Autour des nombres premiers . . . 65

Généralités et syntaxe des structures de contrôles 67
Fonctions, bibliothèques 71
Les listes et les tableaux 73
Les possibilités graphiques de la bibliothèque matplotlib 75
Algorithme de Kaprekar 89
Problème du cavalier d'Euler 95

2 Structure de piles 103
I Généralités 103
II Implémentation avec des listes 104
III Exemple d'utilisation des piles 105
III.1 Parenthésage d'une expression 105
III.2 Branchement dans les L-system 106
Exercices 109

3 Récursivité 113
I Fonctions récursives 113
II Étude théorique d'une fonction récursive 117
II.1 Terminaison 117
II.2 Correction 117
II.3 Complexité 117
II.4 Amélioration d'un algorithme récursif 118
III Exemple : algorithme itératif du triangle de Pascal 120
IV Exemple Exponentiation rapide 122
Exercices 125

4 Algorithmes de tris 127
I Algorithme de tri par sélection 128
II Tri par insertion 130
III Tri fusion 133
IV Tri rapide 138
V Comparatif des algorithmes de tris 141
VI Recherche de la médiane 142

Théorie des algorithmes
Fonctionnement de Python
Programmer en Python
Algorithmes à connaître
Ingénieurie numérique
Généralités et syntaxe des structures de contrôles
Fonctions, bibliothèques
Les listes et les tableaux
Les possibilités graphiques de la bibliothèque matplotlib
Algorithme de Kaprekar
Problème du cavalier d'Euler

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 :

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

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

L'invariant de boucle est :

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

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

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

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` :

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

> **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 :

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

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 :

$$\max\left(L[0],L[1],\dots,L[n-j-1]\right) \leqslant L[n-j] \leqslant \cdots \leqslant L[n-2] \leqslant L[n-1]$$

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).

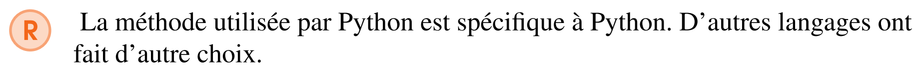
La méthode utilisée par Python consiste à séparer :

- 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 :

```python
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.

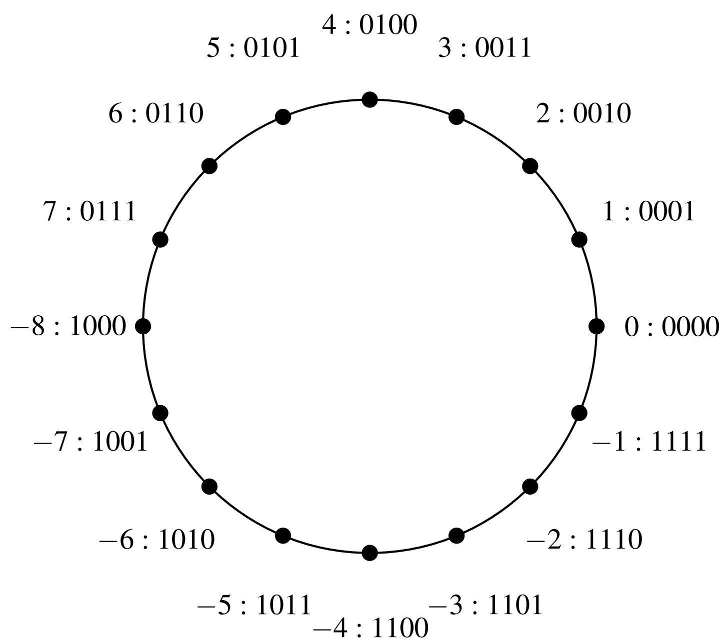
Pour représenter un entier naturel $x \in [[-2^{n-1}, 2^{n-1}-1]]$ sur n bits on procède ainsi :
- Si x est positif ou nul, on le représente comme un entier naturel (sur n bits),
- si x est strictement négatif, **on le représente comme** $2^n + x$ (qui est donc un entier de $[[2^{n-1}, 2^n - 1]]$).

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 10e 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 1001001111000011100000000000000000000000000000000000.`

- 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 :

$$\underline{1.1001001111000011100000000000000000000000000000000000}$$

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 :

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

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

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

est donc correcte, tandis que la formule

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

est grossièrement fausse !

Cela provient des erreurs d'arrondis :

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

R 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

R 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 $[x_0, \ldots, x_{n-1}]$ (souvent des points équirépartis) et la liste des ordonnées $[f(x_1), \ldots, f(x_n)]$.

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.

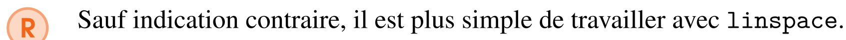
Pour créer les listes de points équirépartis, on dispose de deux fonctions à connaître :

- `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 $[\texttt{xmin}, \texttt{xmax}]$, 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`.

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.

R 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)
```

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

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** : $c = \dfrac{a+b}{2}$. 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
```

■

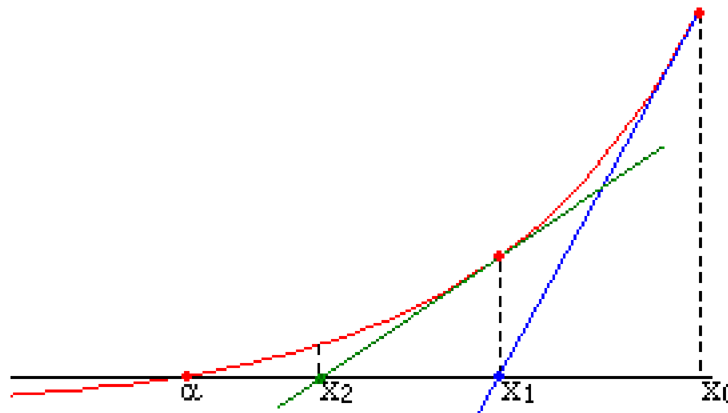
✯ Algorithme de Newton

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 :

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

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 :

$$\begin{aligned}&Y_0 = 0 \\ &\forall i \in [\![0,n-1]\!],\ Y_{i+1} = Y_i + hY_{i+1} \text{ ie } Y_{i+1} = \frac{1}{1-h}Y_i\end{aligned}$$

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 :

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

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 :

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

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 nombres 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` : $(x_0, x_1, \dots, x_{n-1})$,
- la liste des ordonnées `listeY` : $(y_0, y_1, \dots, y_{n-1})$

Les n points

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

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 [xmin, 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 : $x \mapsto \sin(x)$ $\quad x \mapsto x$ $\quad x \mapsto x - \frac{x^3}{6}$ $\quad x \mapsto x - \frac{x^3}{6} + \frac{x^5}{120}$

dessin 2 : $x \mapsto e^x$ $\quad x \mapsto 1+x$ $\quad x \mapsto 1+x+\frac{x^2}{2}$ $\quad x \mapsto 1+x+\frac{x^2}{2}+\frac{x^3}{6}$

dessin 3 : $x \mapsto \ln(1+x)$ $\quad x \mapsto x$ $\quad x \mapsto x - \frac{x^2}{2}$ $\quad x \mapsto x - \frac{x^2}{2} + \frac{x^3}{3}$

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

def DLexp():
    print("-"*10+"fonction exponentiel"+"-"*10)
    xmin = -10
    ymin = -0.2
    xmax = 10
    ymax= 10
    x = linspace(xmin,xmax,100)

    plot(x,exp(x), label= "exponentielle")
    plot(x,1+x, label = "DL 1")
    plot(x,1+x+(x**2)/2, label = "DL 2")
    plot(x,1+x+(x**2)/2 + (x**3)/6, label = "DL 3")

    axis([xmin, xmax, ymin, ymax])
    plot([xmin,xmax],[0,0], color='black')
    plot([0,0],[ymin,ymax], color='black')

    title("fonction exponentielle et ses DLs")
    xlabel("x")
    ylabel("y")

    legend()

    show()
    return

def DLln():
    print("-"*10+"x -> ln(1+x)"+"-"*10)
    xmin = -0.1
    ymin = -20
    xmax = 5
    ymax= 10
    x = linspace(10**(-10),xmax,100)

    plot(x,1+log(x), label="ln")
    plot(x,x, label= "DL 1" )
    plot(x,x-(x**2)/2, label = "DL 2")
    plot(x,x-(x**2)/2 + (x**3)/3, label = "DL 3")

    axis([xmin, xmax, ymin, ymax])
    plot([xmin,xmax],[0,0], color='black')
    plot([0,0],[ymin,ymax], color='black')

    title("fonction logarithme et ses DLs")
    xlabel("x")
    ylabel("y")

    legend()

    show()
    return

```

```
def courbeParam1() :
    t = linspace(0,2*pi,200)
    x = (cos(t))**3
    y = (sin(t))**3
    plot(x,y)
    show()

def courbeParam2() :
    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 $(x_0, y_0), \dots, (x_{n-1}, y_{n-1})$ 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 $i = 0 \dots n-1$
   - $(X_i, Y_i) = \frac{1}{2}(y_i, x_i)$,
   - $(X_{n+i}, Y_{n+i}) = \frac{1}{2}(x_i, 1 + y_i)$,
   - $(X_{2n+i}, Y_{2n+i}) = \frac{1}{2}(1 + x_i, 1 + y_i)$,
   - $(X_{3n+i}, Y_{3n+i}) = \frac{1}{2}(2 - y_i, 1 - x_i)$.
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 $c \notin \mathscr{M}$.

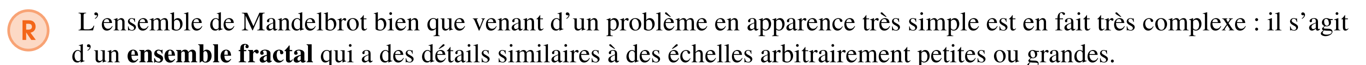
1. Écrire une fonction `suiteIter` qui prend en entrée : un entier `niterMax` et un complexe `c`. Cette fonction construit les `niterMax` premiers itérées et renvoie la variable `diverge`.
   Cette variable booléenne contiendra `True` si au cours des `niterMax` premières itérations, le module de `z` 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
"""

## import
from pylab import *

## fonctions
def suiteIter(niterMax, c):
    """
    entrée: niterMax = int
            = maximum des itérations
            c = complexe
            = paramètre de la suite
    sortie: diverge = bool
            = False si au bout de niterMax itérations, la suite est toujours de module
    inférieur à 2 true sinon

    """
    u = 0
    i = 0
    diverge = False
    while not(diverge) and i <= niterMax :
        u=u**2 + c
        i += 1
        if abs(u)>2 :
            diverge = True

    return diverge

## code
m = 200
img = zeros( (m,m))
x = linspace(-2, 0.5, m)
y = linspace(-1.5, 1.25, m)
for i in range(m):
    for j in range(m):
        c = x[i] + y[j]*1J
        if niter( 100, c) :
            img[i,j] = 255
        else:
            img[i,j] = 0
matshow(img)
show()
```



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()
```

Algorithme de Kaprekar

Pelletier Sylvain PSI, LMSC

En mathématiques, l'algorithme de Kaprekar est un algorithme qui transforme un nombre entier en un autre, de façon répétitive jusqu'à arriver à un cycle. Il fut découvert en 1949 par le mathématicien indien Dattatreya Ramachandra Kaprekar pour les nombres de quatre chiffres, mais il peut être généralisé à tous les nombres.

L'algorithme de Kaprekar consiste à associer à un nombre quelconque n un autre nombre $K(n)$ généré de la façon suivante :

- on considère un nombre n, on le décompose comme la liste de ses chiffres ;
- on forme le nombre n_1 en arrangeant les chiffres du nombre n dans l'ordre croissant et le nombre n_2 en les arrangeant dans l'ordre décroissant ;
- on pose $K(n) = n_2 - n_1$;
- on itère ensuite le processus avec $K(n)$.

Exemple V.12 En partant du nombre 5294, on obtient $K(5294) = 9542 - 2459 = 7083$. En répétant le processus, $K(7083) = 8730 - 378 = 8352$. Puis, $K(8352) = 6174$. On constate que $K(6174) = 6174$ et que l'algorithme conduit alors à un nombre fixe.

Si on commence avec 634, on obtient successivement 297, 693, 594, 495, 495, etc. On obtient là aussi un nombre qui ne varie plus.

Avec 52, la séquence est la suivante : 52, 27, 45, 09, 81, 63, 27...

Partant de 63954, on obtient 63954, 61974, 82962, 75933, 63954, 61974, etc. La séquence se répète. ■

Le but de ce TD est d'écrire l'algorithme de Kaprekar. On commencera par les nombres à 4 chiffres pour lesquels on écrira un algorithme de tri, puis on généralisera à des nombres de taille quelconque.

Vous disposez d'un script qui guide votre travail. Commentez / décommentez au fur et à mesure ce script et testez soigneusement chaque étape.

☆ Kaprekar à quatre chiffres

Étape 1 Écrire une fonction `decompose` qui prend en entrée un nombre n (de taille quelconque) et calcule la liste de ses chiffres. Ainsi, `decompose(523)` est `[5,2,3]`

NB : vous devez écrire l'algorithme « à la main » et non utiliser `str`.

Étape 2 Écrire une fonction `recompose` qui effectue l'opération inverse : à partir de la liste (de taille quelconque) des chiffres renvoie l'entier n. Ainsi, `recompose([5,2,3])` est `523`

Indication : si on note $L_0, \dots, L_{p-1}$ les éléments de L, alors on a :

$$n = \sum_{i=0}^{p-1} L_i 10^{p-i-1}$$

Étape 3 Écrire une fonction `tri4` qui prend en entrée une liste `L` de 4 nombres, et qui sort la liste `Lt` qui contient les éléments de `L` dans l'ordre croissant.

Pour cela :

- Appliquer l'algorithme vu en cours pour trouver les valeurs de `[mini, imini, maxi, imaxi]` qui sont les valeurs et positions du minimum et maximum, (si il y en a plusieurs, peu importe lequel est choisi).
- Créer une liste `elmtRestant` contenant deux éléments (a,b) qui ne sont ni maximum ni minimum.
- Renvoyer la liste `[mini, a, b, maxi]` ou la liste `[mini, b, a, maxi]` selon les cas.

D'autres possibilités existent pour cette étape.
Le script contient un test sur 50 listes aléatoires pour vérification.

Étape 4 Écrire une fonction `KaprekarSuivant` qui prend en entrée un nombre n à quatre chiffres et qui retourne $K(n)$. Tester.

Étape 5 Modifier le script pour effectuer l'opération suivante :

- on part d'un nombre à quatre chiffres (écrit dans le script),
- on affiche la suite des nombres de Kaprekar jusqu'à avoir 6174.

Tester soigneusement avec différentes valeurs.

R On peut montrer que toute suite générée par l'algorithme de Kaprekar à partir d'un nombre de quatre chiffres non tous égaux se stabilise sur 6174 et donc cet algorithme termine.

✯ Kaprekar de taille quelconque

Étape 6 On souhaite tester cet algorithme sur des nombres de taille quelconque.

- Pour trier une liste de taille quelconque, on va utiliser la fonction `sorted`. La syntaxe est simplement `Lt=sorted(L)`. Le tri est par ordre croissant.
  Modifier la fonction `KaprekarSuivant` pour utiliser `sorted` à la place de `tri4`.
- L'autre difficulté est qu'il n'y a pas un point fixe, mais des cycles, c'est-à-dire une séquence qui se répète.
  On propose donc de stocker les valeurs de la suite des nombres de Kaprekar dans une liste et d'arrêter lorsqu'une itération est déjà dans la liste. On a alors un cycle. Par exemple : 63954, 61974, 82962, 75933, 63954 est un cycle de longueur 5.
  Modifier l'algorithme pour qu'il affiche les valeurs de la suite de Kaprekar jusqu'à avoir un cycle.
- Ajouter l'affichage du cycle obtenu et de sa longueur.

✯ Kaprekar avec tri par sélection

Étape 7 On souhaite écrire l'algorithme de tri sans utiliser `sorted`. Pour cela, on peut utiliser un tri par sélection :

- Étant donné une liste `L` quelconque, on cherche le minimum de cette liste et sa position `mini,imini`.
- On le place en première position avec l'opération :

```
L[imini], L[0] = L[0] , L[imini]
```

- On cherche le minimum de la liste `L` pour les éléments de 1 au dernier et la position du minimum. On place ce minimum en position 1 avec l'opération :

```
L[imini], L[1] = L[1] , L[imini]
```

- Ainsi, pour $k \in [\![0, n-2]\!]$: on cherche le minimum de L pour les éléments k au dernier et la position de ce minimum, puis on place ce minimum en position k.

Correction : Version pour les chiffres de longueur 4 :

```
## modules importés
###################
from random import randrange


### fonctions:
##############
def decompose(n):
    """
    entrée: n = int = entier à décomposer
    sortie: L = list = liste des chiffres de n
    """
    L = []
    while n >0 :
        L.insert(0, n%10)
        n = n // 10
    return L

def recompose(L):
    """
    entrée: L = list = liste des chiffres de n
    sortie: n = int = entier reconstuit
            à partir de L
    """
    n = 0
    p = len(L)
    for i in range(p) :
        n += 10**(p-i-1) * L[i]
    return n

```

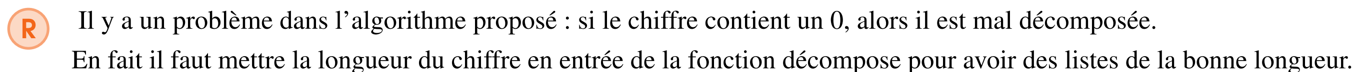
```
def tri4(L):
    """
    entrée: L = list de longueur 4
              = liste à trier
    sortie: Lt = liste de longueur 4
              = les éléments  de L classés par ordre croissant
    """
    mini = maxi = L[0]
    imini = imaxi = 0
    for i in range(len(L)):
        if L[i] > maxi :
            imaxi, maxi = i, L[i]
        elif L[i] < mini :
            imini, mini = i, L[i]

    deuxieme, troisieme = 9, 0
    for i in range(4):
        if i not in [imini, imaxi] and L[i]<= deuxieme :
            deuxieme = L[i]
        if i not in [imini, imaxi] and L[i]>= troisieme :
            troisieme = L[i]
    return [ mini, deuxieme, troisieme, maxi]

    # autre possibilité:

    # elmtRestant = []
    # for i in range(4):
    #     if i not in [imini, imaxi] :
    #         elmtRestant.append(L[i])

    # ou encore:
    # elmtRestant = [L[i] for i in range(4) if i not in [imini, imaxi]

    # a,b = elmtRestant
    # if a < b :
    #     return [mini, a, b, maxi]
    # else :
    #     return [mini, b, a, maxi]



def KaprekarSuivant(n):
    """
    entrée: n = int = entier
    sortie: Kn = int = terme obtenu à partir de n par l'algo de Kaprekar
    """
    L = decompose(n)
    Lt = tri4(L)
    n1 = recompose(Lt)
    n2 = recompose([ Lt[3], Lt[2], Lt[1], Lt[0] ])
    return n2 - n1



### script
##########
print("---algorithme de Kaprekar---")

## test compose / recompose
#print("décomposition de 14567: ",decompose(14567))
#print("recomposition de [7,2,4,5,6,3]", recompose([7,2,4,5,6,3]))
```

```
# test sur 150 entiers au hasard
for i in range(150):
    alea = randrange(10000)
    result = recompose(decompose(alea))
    if alea != result :
        print("errreur dans recompose / decompose",alea, decompose(alea), result)

## test tri4
#for L in [ [1,4,2,3], [1,1,2,4], [2,3,4,1], [1,7,4,4] ]:
#    print("tri de ",L,tri4(L))

# for i in range(50): # 50 tests
#     # création d'une liste de quatre éléments aléatoires de [|0,10|]:
#     L = [ randrange(10) for j in range(4) ]
#     if tri4(L) != sorted(L) :
#         print("erreur de tri: ",L, tri4(L))
# print("pas d'errreur de tri")


## test KaprekarSuivant
#n = 5294
#print("terme suivant: ", KaprekarSuivant(5294) )


## script principal:
n = 1965
# à vous d'écrire la suite
while n != 6174 :
    print(n)
    n = KaprekarSuivant(n)
print("on atteint la valeur point fixe 6174")

print("---fin de l'algorithme de Kaprekar---")
```

Version pour les chiffres de longueur quelconque :

```
### fonctions:
##############
def decompose(n):
    """
    entrée: n = int = entier à décomposer
    sortie: L = list = liste des chiffres de n
    """
    L = []
    while n >0 :
        L.insert(0, n%10)
        n = n // 10
    return L

def recompose(L):
    """
    entrée: L = list = liste des chiffres de n
    sortie: n = int = entier reconstuit
            à partir de L
    """
    n = 0
    p = len(L)
    for i in range(p) :
        n += 10**(p-i-1) * L[i]
    return n


def tri(L):
```

```python
    """
    entrée: L = list
              = liste à trier
    sortie: Lt = liste
              = les éléments  de L classés par ordre croissant
    """
    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




def KaprekarSuivant(n):
    """
    entrée: n = int = entier
    sortie: Kn = int = terme obtenu à partir de n par l'algo de Kaprekar
    """
    L = decompose(n)
    Lt = tri(L)
#    Lt = sorted(L)
    n1 = recompose(Lt)
    n2 = recompose(Lt[::-1])
    return n2 - n1




### script
##########
## script principal:
n = 1965145
listIter = []
while n not in listIter :
    print(n)
    listIter.append(n)
    n = KaprekarSuivant(n)

# recherche de n dans listIter
# c'est la recherche d'un nombre dans une liste
i = 0
while listIter[i] != n :
    i += 1
# la boucle se termine puisque n est dans listIter
print("cycle trouvé :", listIter[i:], " longueur: ", len(listIter) -i )
```


Il y a un problème dans l'algorithme proposé : si le chiffre contient un 0, alors il est mal décomposée.
En fait il faut mettre la longueur du chiffre en entrée de la fonction décompose pour avoir des listes de la bonne longueur.

Problème du cavalier d'Euler

Pelletier Sylvain PSI, LMSC

On se propose de résoudre le problème suivant :

Trouver la trajectoire d'un cavalier sur l'échiquier, posé sur une case de départ quelconque, pour qu'il passe une et une seule fois par toutes les cases de l'échiquier.

Ce problème est connu sous le nom de **problème du cavalier d'Euler**. Pour une étude complète du cavalier d'Euler, on pourra consulter l'article wikipedia :
`https://fr.wikipedia.org/wiki/Probl%C3%A8me_du_cavalier`.

On rappelle qu'un cavalier se déplace de deux cases dans une direction et d'une case dans une autre (exemple : deux fois au nord et une fois à l'ouest, ou deux fois à l'est et une fois au sud, etc.). Un cavalier placé au centre de l'échiquier peut ainsi se déplacer sur 8 cases, mais un cavalier situé dans un coin n'aura que deux déplacements possibles.

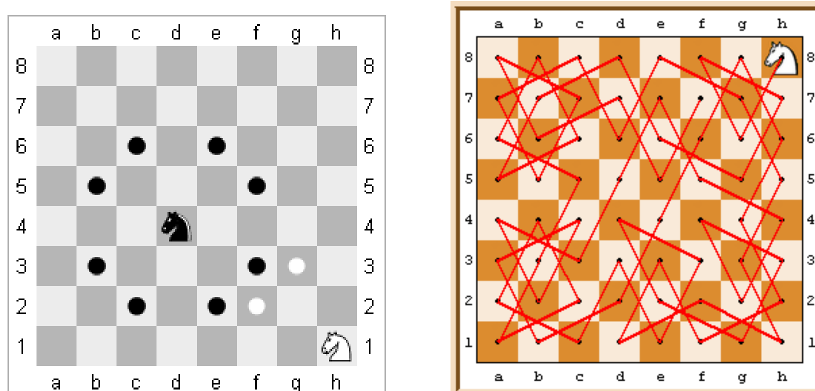


FIGURE 1.1 – Déplacement du cavalier et une solution du problème d'Euler

Une méthode heuristique pour résoudre ce problème consiste étant donné une position du cavalier à

- déterminer la liste des coups possibles depuis cette position,
- pour chacun de ces coups possibles, déterminer combien de coups sont possibles depuis cette nouvelle position.
- On choisit alors le coup qui possède le minimum de déplacements possibles.

Le but de ce projet est donc de programmer cette méthode.

Bien sûr le but n'est pas uniquement de résoudre un problème d'échecs. On peut reformuler le problème ainsi : étant donné un graphe, on veut visiter tous les noeuds du graphe en passant une et une seule fois par chaque noeud. Ce problème est très complexe.

La méthode proposée est une heuristique, rien ne prouve qu'elle donne le bon résultat. Il arrive qu'elle échoue.

La première étape est de définir précisément la **structure de données** dont on a besoin pour ce projet, c'est-à-dire comment on va représenter les données dans la machine.

Pour l'échiquier, on va utiliser un `array` 8×8 appelé `jeu`, qui représentera l'état des cases de l'échiquier.

Au début, toutes les cases ont la valeur `NON_VISITEE` que l'on prendra égale à 0, sauf la case de départ qui contient la valeur 1. Lorsque le cavalier est placé sur une case, celle-ci prend la valeur du numéro du coup. En affichant (avec `print`) cet `array`, on aura la trajectoire du cavalier.

Une case de l'échiquier sera représentée par un couple $(i, j) \in [\![0, 7]\!]^2$. On aura aussi besoin d'une structure du type : « position et nombre de coups possibles depuis cette position ». Ce sera donc un triplet (i, j, k) où (i, j) est une case et k un entier donnant le nombre de coups possibles depuis (i, j).

■ **Exemple V.13** Voici un exemple de solution (affiché avec Python) :

```
[[ 29.   12.   33.   46.   35.   14.   63.   58.]
 [ 32.   25.   30.   13.   64.   57.   36.   15.]
 [ 11.   28.   47.   34.   45.   54.   59.   62.]
 [ 26.   31.   24.   53.   56.   61.   16.   37.]
```

```
[ 23.  10.  27.  48.   1.  44.  55.  60.]
[  4.   7.  20.  41.  52.  49.  38.  17.]
[  9.  22.   5.   2.  19.  40.  43.  50.]
[  6.   3.   8.  21.  42.  51.  18.  39.]]
```

■

⋆ Analyse descendante de l'algorithme

Pour résoudre le problème, on commence par le découper en sous-problèmes plus simples. Chacun de ces sous-algorithmes correspond à une fonction que vous devez écrire.

Pour chaque fonction écrite, il faut prévoir des tests précis permettant de vérifier qu'il n'y a pas d'erreurs. Laissez dans le code, en commentaire, les tests que vous avez choisis pour tester chaque fonction.

Voici le découpage proposé :

- Une fonction `cases1move(jeu,i,j)` détermine la liste des déplacements possibles à partir d'une position `i,j` du cavalier et du `jeu`. La sortie est une liste de cases.
  L'entête de cette fonction est :

```
def cases1move(jeu,i,j) :
    """
    entrée:
    jeu = array 8x8
        = état actuel de l'échiquier
    (i,j) =  case = couple de int [|0,7|]
          = position actuelle du cavalier
    sortie:
    lc = liste de cases
       = liste des cases que l'on peut atteindre
          à partir de [i,j]
    """
```

  Le déplacement `(i,j)` vers `(i0,j0)` n'est possible que si la case `(i0,j0)` existe et n'a pas encore été atteinte.
- Une fonction `casesPoids(jeu,i,j)` calcule la liste des déplacements possibles et le nombre de déplacements possibles depuis chaque nouvelle position à partir des mêmes entrées que `case1move`.
  La sortie est une liste de la forme `[i1,j1,k]`, où $(i1, j1)$ est une case que l'on peut atteindre à partir de (i, j), et k est le nombre de cases que l'on peut atteindre à partir de $(i1, j1)$ (y compris (i, j)).
  Bien entendu, cette fonction utilise la fonction `cases1move`.
  L'entête de cette fonction est :

```
def casesPoids(jeu,i,j) :
    """
    entrée:
    jeu = array 8x8
        = état actuel de l'échiquier
    (i,j) =  case = couple de int [|0,7|]
          = position actuelle du cavalier
    sortie:
    lcp = liste de triplets [i1,j1,k]
         = [i1,j1] peut être atteinte depuis [i,j]
         k nbr de cases que l'on peut atteindre
         à partir de [i1,j1]
    """
```

- Une fonction `calculeMin3(l3uplet)` qui prend en entrée une liste `l3uplet` de 3-uplet et qui renvoie l'indice `l` du 3uplet de `l3uplet` dont le troisième élément est minimal.
  Autrement dit, on note `i0, j0, n=l3uplet[l]`, et `n` vérifie :

$$n = \min\left\{c \middle| (a,b,c) \in \texttt{l3uplet}\right\}$$

  L'entête est :

```
def calculeMin(l3uplet) :
    """
    entrée: l3uplet = liste de 3uplet
    sortie: l = int = indice du 3uplet
      dont le troisième élément est minimal
    """
```

On a alors simplement un calcul de minimum.
NB : interdit d'utiliser une fonction `min`, vous devez refaire « à la main ».

- Une fonction `coupSuivant(jeu,i,j)` détermine le coup suivant à partir de la position `i,j` du cavalier. La sortie est un couple $(i1, j1)$ donnant la nouvelle position choisie du cavalier. Cette fonction utilise les fonctions précédentes. On convient que la sortie est $(-1,-1)$ si aucun coup n'est possible.
  L'entête est :

```
def coupsSuivant(jeu,i,j) :
    """
    entrée:
    jeu = array 8x8
        = état actuel de l'échiquier
    (i,j) =  case = couple de int [|0,7|]
          = position actuelle du cavalier
    sortie:
    (i1,j1) = case
      = position suivante du cavalier
      (-1,-1) si pas de position possible
    """
```

- Enfin, le reste des instructions est écrite directement dans le script pour avoir la trajectoire du cavalier. Il faut initialiser l'échiquier puis poser le cavalier sur une case, et déplacer le cavalier de case en case, jusqu'à avoir rempli l'échiquier ou ne plus avoir de solution.

Correction : Voici un exemple de correction.

```
# -*- coding: utf-8 -*-

"""
Résolution du  problème du cavalier en Python
"""

from numpy import zeros, array


# constante nommées
NON_VISITE = 0

def cases1move(jeu,i,j) :
    """
    entrée:
    jeu = array 8x8
        = état actuel de l'échiquier
    (i,j) =  case = couple de int [|0,7|]
          = position actuelle du cavalier
    sortie:
    lc = liste de cases
       = liste des cases que l'on peut atteindre
          à partir de [i,j]
    """
    lc = []

    if i+2 < 8 and j-1 >= 0  and jeu[i+2, j-1]==NON_VISITE :
        lc.append([i+2,j-1])
    if i+2 < 8 and j+1 < 8  and jeu[i+2, j+1]==NON_VISITE :
        lc.append([i+2,j+1])
    if i+1 < 8 and j-2 >= 0  and jeu[i+1, j-2]==NON_VISITE :
```

```
        lc.append([i+1,j-2])
    if i+1 < 8 and j+2 < 8  and jeu[i+1, j+2]==NON_VISITE :
        lc.append([i+1,j+2])
    if i-1 >= 0 and j-2 >= 0 and jeu[i-1, j-2]==NON_VISITE :
        lc.append([i-1,j-2])
    if i-1 >= 0 and j+2 < 8 and jeu[i-1, j+2]==NON_VISITE :
        lc.append([i-1,j+2])
    if i-2 >= 0 and j-1 >= 0 and jeu[i-2, j-1]==NON_VISITE :
        lc.append([i-2,j-1])
    if i-2 >= 0 and j+1 < 8 and jeu[i-2, j+1]==NON_VISITE :
        lc.append([i-2,j+1])
    return lc


def casesPoids(jeu,i,j) :
    """
    entrée:
    jeu = array 8x8
        = état actuel de l'échiquier
    (i,j) =  case = couple de int [|0,7|]
            = position actuelle du cavalier
    sortie:
    lcp = liste de triplets [i1,j1,k]
         = [i1,j1] peut être atteinte depuis [i,j]
         k nbr de cases que l'on peut atteindre
         à partir de [i1,j1]
    """
    lcp = []
    lc = cases1move(jeu,i,j)
    for i1,j1 in lc :
        lc2 = cases1move(jeu,i1,j1)
        k = len(lc2)
        lcp.append([i1,j1,k])
    return lcp



def calculeMin(l3uplet) :
    """
    entrée: l3uplet = liste de 3uplet
    sortie: l = int = indice du 3uplet
      dont le troisième élément est minimal
    """
    l = 0
    mini = l3uplet[0][2]
    n = len(l3uplet)
    for i in range(1,n) :
        if l3uplet[i][2] < mini :
            l = i
            mini = l3uplet[i][2]
    return l


def coupsSuivant(jeu,i,j) :
    """
    entrée:
    jeu = array 8x8
        = état actuel de l'échiquier
    (i,j) =  case = couple de int [|0,7|]
            = position actuelle du cavalier
    sortie:
    (i1,j1) = case
      = position suivante du cavalier
```

```
    (-1,-1) si pas de position possible
    """
    l3 = casesPoids(jeu,i,j)
    if l3 == [] :
        print("erreur: bloqué à la case ",i,j)
        return -1,-1
    imd = calculeMin(l3) #indice du meilleur coups
    i = l3[imd][0] # les deux composantes du meilleur coups
    j = l3[imd][1]
    return i,j

jeu = zeros( (8,8))

#position départ
i=4
j=4
jeu[i, j]=1

t=2
while i!= -1 and t<=64 :
    print("jeu pour t=:",t)
    print(jeu)
    [i,j]=coupsSuivant(jeu,i,j)
    print("nouvelle position du cavalier: ",i,j)
    jeu[i,j]=t
    t += 1
    input("---stop---")


if t==65 :
  print("pb résolu!")
  print("jeu final:")
  print(jeu)
else :
  print("échec de la méthode")
```

Voici un autre exemple avec une programmation différente.

```
# -*- coding: utf-8 -*-

"""
Résolution du  problème du cavalier en Python
"""

from numpy import zeros, array

# constante nommées
NON_VISITE = 0

def cases1move(jeu,i,j) :
    """
    entrée:
    jeu = array 8x8
        = état actuel de l'échiquier
    (i,j) =  case = couple de int [|0,7|]
          = position actuelle du cavalier
    sortie:
    lc = liste de cases
       = liste des cases que l'on peut atteindre
          à partir de [i,j]
    """
    lc = []
    deplacement = [ [2,1], [2,-1],
                    [1,2], [1,-2],
```

```
                    [-1,2], [-1,-2],
                    [-2,1], [-2,-1] ]
    for di, dj in deplacement :
        if ( 0 <= i+di< 8 and  0 <= j+dj< 8
             and jeu[i+di, j+dj] == NON_VISITE ):
             lc.append([i+di, j+dj])
    return lc


def casesPoids(jeu,i,j) :
    """
    entrée:
    jeu = array 8x8
        = état actuel de l'échiquier
    (i,j) =  case = couple de int [|0,7|]
          = position actuelle du cavalier
    sortie:
    lcp = liste de triplet [i1,j1,k]
         = [i1,j1] peut être atteinte depuis [i,j]
         k nbr de cases que l'on peut atteindre
         à partir de [i1,j1]
    """
    lcp = [ [i1, j1, len(cases1move(jeu,i1,j1))] for i1,j1 in cases1move(jeu,i,j)]
    return lcp

def calculeMin(l3uplet) :
    """
    entrée: l3uplet = liste de 3uplet
    sortie: l = int = indice du 3uplet
      dont le troisième élément est minimal
    """
    l = 0
    mini = l3uplet[0][2]
    n = len(l3uplet)
    for i in range(1,n) :
        if l3uplet[i][2] < mini :
            l = i
            mini = l3uplet[i][2]
    return l


def coupsSuivant(jeu,i,j) :
    """
    entrée:
    jeu = array 8x8
        = état actuel de l'échiquier
    (i,j) =  case = couple de int [|0,7|]
          = position actuelle du cavalier
    sortie:
    (i1,j1) = case
      = position suivante du cavalier
      (-1,-1) si pas de position possible
    """
    l3 = casesPoids(jeu,i,j)
    if l3 == [] :
        print("erreur: bloqué à la case ",i,j)
        return -1,-1
    imd = calculeMin(l3) #indice du meilleur coups
    i = l3[imd][0] # les deux composantes du meilleur coups
    j = l3[imd][1]
    return i,j

jeu = zeros( (8,8))
```

```
#position départ
i=2
j=4
jeu[i,j]=1

t=2
while i!= -1 and t<=64 :
     [i,j]=coupsSuivant(jeu,i,j)
     jeu[i,j]=t
     print("jeu pour t=",t, "le cavalier est en ", i, j)
     print(jeu)
     t += 1
     input("---stop---")

if t==65 :
  print("pb résolu!")
else :
  print("échec de la méthode")
```

Généralités
Implémentation avec des listes
Exemple d'utilisation des piles
Exercices

2 — Structure de piles

I Généralités

Une pile est un conteneur dans lequel on peut uniquement :

- savoir si la pile est vide,
- consulter le dernier élément de la pile,
- enlever le dernier élément de la pile (**dépiler**),
- ajouter un élément *au-dessus* (**empiler**).

On parle de pile `LIFO` (last in first out).

Voici des exemples d'utilisation de piles :

Dans un navigateur d'aide : les pages visités dans la session sont sauvegardés dans une pile. À chaque page consultée, un élément est ajouté à la pile, on peut reculer d'une page (c'est-à-dire dépiler une page).

Dans quasiment tous les logiciels, il existe une fonction `Annuler` : toutes les actions sont sauvegardées dans une pile, et on peut annuler la dernière action « en dépilant ».

C'est très utile dans les mécanismes bas niveaux : vérifier qu'une expression est syntaxiquement correcte, c'est-à-dire par exemple qu'à chaque parenthèse ouvrante corresponds une parenthèse fermante.
C'est particulièrement utile pour les langages marqués comme le `HTML` qui sert à faire des pages web. Pour ces langages, il y a des balises ouvrante de la forme `<a>` et des balises fermantes `</a>` qui délimitent des blocs.
Chaque balise ouvrante doit correspondre à une balise fermante placée avant avec la règle suivante : les blocs délimités par les balises peuvent être imbriquées mais pas se « croiser ».
Autrement dit, on peut avoir :

```
<a>
   <b>
```

```
  </b>
</a>
```

mais pas :

```
<a>
  <b>
</a>
  </b>
```

La vérification syntaxique d’un tel langage se fait en lisant le fichier ligne par ligne et en gardant une pile des balises utilisées :

- quand une nouvelle balise ouvrante est lue, elle est ajouté en haut de la pile,
- lorsqu’une balise fermante est lue, on dépile la dernière balise et on vérifie la correspondance entre la balise dépilée et la balise fermante.
- à la fin du fichier, la pile doit être vide.

On peut aussi vérifier qu’un programme Python est bien écrit en vérifiant qu’à chaque ligne :

- soit une nouvelle indentation est crée,
- soit on revient à un des niveaux d’indentations précédents.

> La principale utilité des piles est la gestion des fonctions récursives que l’on verra au prochaine chapitre.

On utilise souvent des boucles : `while len(pile)>0`. Une fonction de terminaison naturelle est alors la longueur de la pile.

Il existe aussi la **structure de file** dite FIFO (first in first out) qui permet par exemple de gérer une file d’attente à un serveur d’impression, ou une file de données qui doivent être acheminées par un routeur. Cette structure étant hors-programme, on ne l’étudiera pas en détail, néanmoins, il est facile de l’implémenter avec des listes.

II Implémentation avec des listes

> Conformément au programme, on va utiliser des listes pour simuler des files. On utilisera les syntaxes suivantes :
>
> - `pile = []` ou encore `pile = list()` pour créer une pile (ie une liste vide)
> - `len(pile) == 0` pour savoir si la pile est vide,
> - `pile[-1]` donne accès au dernier élément,
> - `pile.pop()` pour enlever et retourner le dernier élément,
> - `pile.append(x)` pour ajouter un nouvel élément `x`.

On peut donc utiliser : `pile.pop()` pour enlever le dernier élément, ou `y=pile.pop()` pour affecter y à la valeur du dernier élément et l’enlever de la pile.

III Exemple d'utilisation des piles

III.1 Parenthésage d'une expression

Considérons une expression constituée de parenthèses `()`, crochet `[]` ou accolade `{}` et d'autres lettres.

On souhaite vérifier que l'expression est bien parenthésée, c'est-à-dire :

- qu'à chaque symbole « ouvrant » corresponds bien un symbole « fermant » placée avant.
- Que les symboles ne se croisent pas.

On construit donc une pile définie ainsi :

- au début la pile est vide,
- à chaque caractère lu, si c'est un caractère ouvrant, on empile ce caractère,
- si c'est un caractère fermant, on vérifie que le dernier caractère empilé lui correspond. Si il n'y a pas d'erreur, on dépile.
- à la fin, on vérifie que la pile est vide.

Voici un exemple de programme que l'on peut écrire ainsi :

```
def bienParenthese(s) :
  """
  entrée: s = chaîne de caractères
            = expression à contrôler
  sortie: booléen = True si bien parenthése,
                    False sinon
  """

  pile = [] # pile des symboles utilisées
  # = une liste de caractères.

  for cara in s :

    # facultatif: passe directement au caractère suivant
    # si cara n'est pas un symbole:
    if cara  not in ("(", ")", "[", "]", "{", "}") :
      continue

    if cara in ( "(", "[", "{" ) :
      pile.append(cara)
    elif cara == ")" :
      if len(pile) == 0 or pile[-1] != "(" :
        return False
      pile.pop()
    elif cara == "]" :
      if len(pile) == 0 or pile[-1] != "[" :
        return False
      pile.pop()
    elif cara == "}" :
      if len(pile) == 0 or pile[-1] != "{" :
        return False
      pile.pop()

  return ( len(pile) == 0 )
```

On peut modifier le programme ci dessus de manière à vérifier que l'ordre des parenthèses est respecté dans le sens où les symboles ouvrants les plus intérieurs doivent être des parenthèses, puis des crochets, puis des accolades.
Pour cela, il faut remplacer la ligne :

```
if cara in ( "(", "[", "{" ) :
  pile.append(cara)
```

Par :

```
if cara == "(" :
  if len(pile) == 0 or pile[-1] == "[" :
    pile.append(cara)
  else :
    return False
elif cara == "[" :
  if len(pile) != 0 and pile[-1] == "("
    pile.append(cara)
  else :
    return False
elif cara == "{" :
  if len(pile) >= 2 and pile[-1] == "["  :
    pile.append(cara)
  else :
    return False
```

III.2 Branchement dans les L-system

Les L-system sont une modélisation fractale de la croissance des plantes.

Le principe est de créer avec un processus récursif une liste d'instructions qui seront exécutées par la machine et qui permettront d'obtenir une représentation de la plante.

Ces instructions sont du type *langage tortue* : on dispose d'un système graphique (*une tortue*) qui est défini par :

- une position $(x,y) \in \mathbb{R}^2$, une direction $\theta \in \mathbb{R}$,
- une couleur $c \in [\![0, \texttt{CMAX}]\!]$ et une épaisseur de trait $t \in \mathbb{N}^*$.

L'instruction dans le langage `L-systeme` est donc une chaîne caractère contenant les symboles :

`+` pour augmenter la valeur de θ d'une constante $\Delta\theta$,
`-` pour diminuer la valeur de θ de $\Delta\theta$,
`F` pour tracer un trait (de longueur 1) à la couleur et l'épaisseur courante. La tortue se déplace alors en traçant, ce qui signifie que les valeurs de (x,y) sont changées en $(x+\cos\theta, y+\sin\theta)$.
`n` pour passer à la couleur suivante si possible,
`p` pour passer à la couleur précédente si possible,
`l` pour augmenter l'épaisseur du trait,
`s` pour diminuer l'épaisseur du trait,
`[` pour créer un embranchement,
`]` pour revenir à l'embranchement précédent.

Pour mener ce projet, il faut utiliser la structure de données suivante :

- La tortue sera représentée sous la forme d'une liste `[x,y,theta,c,t]` donnant les indications sur la position, la direction, la couleur et l'épaisseur du trait.
- on dispose d'une fonction `trace` qui prends en entrée une tortue et sort une tortue (qui s'est déplacée après avoir tracé). Cette fonction modifie le graphique.
- Enfin, pour réaliser les embranchements, on utilisera une pile, qui sera donc une liste de tortue (ie de 5-uplets).

Voici un exemple de réalisation :

```python
def execute(ordre) :
  """
  ordre = chaîne de caractères +-Fnpls[]
    = listes des instructions tortue
  sortie = graphique + True si la chaîne est correcte
                      / False sinon
  """

  # initialisation de la tortue
  # au centre et vers le haut
  # couleur 0 et épaisseur 1
  tortue = [0, 0, pi/2, 0, 1]

  pile = [] #pile de tortues pour les embranchements

  for cara in ordre :
    if cara == "+" :
      tortue[2] += DTHETA
    elif cara == "-" :
      tortue[2] -= DTHETA
    elif cara == "F" :
      tortue = trace(tortue)
    elif cara == "n" and tortue[3] < CMAX :
      tortue[3] += 1
    elif cara == "p" and tortue[3] > 0 :
      tortue[3] -= 1
    elif cara == "l" :
      tortue[4] += 1
    elif cara == "s" and tortue[4] > 0 :
      tortue[4] -= 1
    elif cara == "[" :
      pile.append( tortue.copy() )
    elif cara == "]" :
      if len(pile) == 0 :
        print("pb de parenthésage")
        return False
      tortue = pile.pop()
    else :
        print("instruction incorrectes")
        return False
  return ( len(pile) == 0 )
```

Il faut utiliser `pile.append( tortue.copy() )` et non `pile.append( tortue )`

sinon c'est les références qui sont stockées dans la pile et donc les modifications ultérieures de la liste `tortue` seront exécutées aussi sur les valeurs stockées.

Structure de pile

Pelletier Sylvain — PSI, LMSC

Exercice 1 Palindromes et piles

Un palindrome est une chaîne de caractères qui est identique à elle-même lue en sens inverse : exemple `esse` `radar` est un palindrome mais `math` n'en est pas un.

En utilisant une structure de pile, écrire une fonction `estPalindrome(X)` qui renvoie `True` si la chaîne de caractère `X` est un palindrome, `False` sinon.

Estimer la complexité en espace et en temps.

Correction : Voici un exemple de programme

```
def estPalindrome(X) :
  """
  entrée: X = chaîne de caractères
            = mot à analyser
  sortie: booléen = True si X est palindrome
  """
  pile = []
  n = len(X)
  if n%2 == 0 :
      debut = n//2
      suite = n//2
  else :
      debut = n//2
      suite = n//2+1
  for lettre in X[:debut]:
    pile.append(lettre)
  for lettre in X[suite:]:
    if pile.pop() != lettre :
      return False
  return True



for mot in ("esse", "estse", "math", "estsee", "kayak", "radar") :
  if estPalindrome(mot) :
    print(mot + " est un palindrome")
  else :
    print(mot + " n'est pas un palindrome")
```

Une autre solution qui n'utilise que des piles :

```
def estPalindrome(X) :
  """
  entrée: X = chaîne de caractères
            = mot à analyser
  sortie: booléen = True si X est palindrome
  """
  X = list(X) # convertion en liste
  pile = []
  n = len(X)

  # on enlève de X pour stocker (à l'envers)
  # les n//2 derniers éléments
  for i in range(n//2):
    pile.append(X.pop())
  # on enlève l'élément central si n impair
  if n%2 == 1 :
```

```
        X.pop()

    # on continue à envever dans X
    # mais on dépile en même temps dans pile
    for i in range(n//2):
        if   pile.pop() != X.pop() :
            return False
    # si tout va bien on renvoie True
    return True



for mot in ("esse", "estse", "math", "estsee", "kayak", "radar") :
    if estPalindrome(mot) :
        print(mot + " est un palindrome")
    else :
        print(mot + " n'est pas un palindrome")
```

Exercice 2 Notation polonaise inversée

La **notation polonaise inversée** est la notation post-fixée d'une expression.

Autrement dit, plutôt que d'écrire les opérandes autour de l'opérateur (comme dans $2+3$), on écrit les opérandes puis l'opérateur (`[2,3,"+"]`).

Par exemple, la notation polonaise inverse de l'expression :

$$J = ((13-1)\times 2 + 7/4)$$

est :

```
J = [ 13, 1 , "-" , 2 , "*", 7 , 4 , "/" , "+"]
```

Le gros avantage de cette notation est qu'on n'a pas besoin de parenthèse !

En pratique, c'est celle qui est utilisée dans les mécanismes bas-niveaux pour évaluer les expressions.

En utilisant une pile, créer une fonction `evalueNPI` qui permet d'évaluer une expression écrite en notation polonaise inversée. On utilisera en entrée une liste contenant :

- des nombres (flottants ou entiers) pour les opérandes,
- des caractères `"+"`,`"*"`,`"/"`,`"-"` pour les opérateurs.

Correction :

```
def evalueNPI(expression):
    """
    entrée: expression = liste de (float ou  int)/string
    = expression à évaluer
    sortie: out = float = résultat
    """
    pile = []
    for symbole in expression :
        # on peut aussi faire 4 if.
        if symbole in ( "+", "-", "*", "/" ) :
            opr = pile.pop() # opérande de droite
            opl = pile.pop() # opérande de gauche
            pile.append(eval( str(opl) + symbole + str(opr)))
        else :
            pile.append(symbole)
    out = pile.pop()
    # on vérifie que la pile est vide:
    assert len(pile) == 0
    return out


J =  [ 13, 1 , "-" , 2 , "*", 7 , 4 , "/" , "+"]
```

```
print("l'évaluation de ",J, " est ", evalueNPI(J))

J =  [ 1, 2, "+", 3, "*"]
print("l'évaluation de ",J, " est ", evalueNPI(J))

J =  [ 1,2,"*",3,"+"]
print("l'évaluation de ",J, " est ", evalueNPI(J))

J =  [ 1,2, "+", 3, "+"]
print("l'évaluation de ",J, " est ", evalueNPI(J))
```

Exercice 3 Autour des mélanges de cartes

On souhaite simuler le mélange de carte suivant à l'aide de pile :

- on prends le paquet dans la main, on coupe au hasard, on a donc deux paquets (un dans chaque main)
- on mélange les deux paquets en choisissant au hasard une carte de la main gauche ou une carte de la main droite jusqu'à épuisement des deux paquets.

(c'est le mélange américain)

Écrire une fonction `couper` qui prend une pile en entrée, et la « coupe » en enlevant de son sommet un nombre aléatoire d'éléments qui sont renvoyés dans une seconde pile.

Exemple : si la pile initiale est `[1, 2, 3, 4, 5]`on tire au hasard un entier de $[\![1,4]\!]$ si on obtient 2, alors on renvoie `[4,5]` (les deux éléments situés au-dessus) et la pile initiale devient `[1, 2, 3]`.

Écrire une fonction `mélange` qui prends en entrée deux piles et construit la pile mélangée.

Ce mélange permet le **tour de magie de Gilbreath** :

- on construit un jeu de 32 cartes en alternant une carte rouge / une carte noire (vous utiliserez une liste de 0,1).
- On mélange en introduisant un petit changement : lorsqu'on coupe, on change l'ordre du deuxième paquet. Sur l'exemple ci-dessus, la pile retournée par la fonction `couper` est `[5,4]`.
- Le paquet obtenu contient alors toujours des couples de cartes rouges / noires.

Vérifiez ce principe.

Généraliser au cas d'un jeu de n cartes constitué de groupes de k cartes.

Correction :

```
## bilbiothèque
from pylab import rand

def couper(paquet) :
    """
    entrée: paquet = list = paquet de cartes à couper
    sortie: paquetExtrait = list = paquet "sorti par le haut"
            + modif de paquet
    """
    n = len(paquet)
    nbrCarteEnleve = int ( rand() * (n-1) ) + 1
    paquetExtrait = []
    for i in range(nbrCarteEnleve) :
        paquetExtrait.append( paquet.pop() )
    return paquetExtrait

def melange(paquet1, paquet2):
    """
    entrée: paquet1, paquet2 = liste = deux paquets de cartes.
    sortie: jeuMelange = liste = jeu obtenu en mélangeant alternativement
      paquet1 et paquet2
    """
    jeuMelange = []
    while len(paquet1) != 0 and len(paquet2) != 0 :
        alea = rand()
        if alea < 0.5 :
            jeuMelange.append( paquet1.pop() )
        else :
```

```
            jeuMelange.append( paquet2.pop() )
    if len(paquet1) == 0 :
        jeuMelange.extend(paquet2)
    elif len(paquet2) == 0  :
        jeuMelange.extend(paquet1)
    return jeuMelange

for test in range(15):
    carte = [0,1] * 20
    paquetExtrait = couper(carte)
    jeu = melange(carte, paquetExtrait)
    print(jeu)
```

Fonctions récursives
Étude théorique d'une fonction récursive
Exemple : algorithme itératif du triangle de Pascal
Exemple Exponentiation rapide
Exercices

3 — Récursivité

I Fonctions récursives

Définition I.1 Une fonction est récursive si elle contient une instruction qui l'appelle elle-même.

■ **Exemple I.1** La **fonction factorielle** est le prototype de fonction récursive :

```
def fac(n):
  if n == 0 :
    return 1
  return n * fac(n-1)
```

■

■ **Exemple I.2** L'autre exemple basique est le calcul d'une suite récursive. Soit (u_n) la suite définie par :

$$u_0 = 2 \qquad \text{et} \qquad \forall n \in \mathbb{N},\ u_{n+1} = \frac{1}{3}(u_n + u_n^2)$$

Pour écrire l'algorithme correspondant, on écrit : $\forall n \in \mathbb{N}^*,\ u_n = \frac{1}{3}(u_{n-1} + u_{n-1}^2)$ et on a :

```
def u(n) :
  if n == 0
    return 2
  return (1/3) * ( u(n-1) + u(n-1)**2 )
```

■

L'écriture d'une fonction récursive est ainsi intiment liée au **principe de récurrence** :

- on donne les premiers termes (cas particuliers)
- on indique comment se ramener à l'hypothèse de récurrence (appel à la fonction sur des valeurs inférieures).

L'utilisation de fonction récursive permet généralement d'écrire des algorithmes élégant, court et très proche de la modélisation mathématique. Néanmoins, ces algorithmes ne sont pas toujours efficaces du point de vu de la complexité en temps et en espace. L'étude théorique d'une fonction récursive est plus compliqué que l'étude d'une fonction non récursive.

R Une fonction récursive contient systématiquement plusieurs `return`.

⋆ Pile d'appel d'une fonction récursive

Pour utiliser une fonction récursive, le processeur crée une pile d'appel à la fonction. Par exemple, lorsqu'on calcule `fac(4)` avec `fac` la fonction factorielle, le processeur :

- crée un appel la fonction avec $n = 4$, dans lequel il rencontre le nom `fac(3)`.
- Il crée alors un nouvel appel à la fonction dans un niveau d'exécution inférieur, d'après le principe de portée des variables.
  Il fait donc appel à la fonction avec pour variable d'entrée $n = 3$,
- Il recommence ensuite pour $n = 2$ et $n = 1$.
- Lorsque la fonction se termine (appelée sur l'entrée $n = 1$), la valeur est envoyé à l'espace de nom du niveau d'exécution au-dessus (correspondant donc à l'appel avec $n = 2$), après multiplication par 2, cette fonction se termine et retourne une valeur.
- Cette valeur est aussi transmise au niveau d'exécution situé au-dessus, et ainsi de suite, jusqu'au niveau d'exécution situé au sommet.

La taille de cette pile est liée au **nombre d'appels à la fonction**. En Python, cette pile a une taille limitée à 1000. On obtient l'erreur : `RuntimeError: maximum recursion depth exceeded in comparison` On peut changer cette limite en utilisant la fonction `setrecursionlimit` du module `sys`.

Pour mesurer le nombre d'appel, on pourra utiliser une variable globale :

```
nbrAppel = 0
def fac(n):
  global nbrAppel
  nbrAppel += 1
  if n == 0 :
    return 1
  return n * fac(n-1)
fac(10)
print(nbrAppel)
```

⋆ Exemples d'algorithme itératif écrit de manière récursive

■ Exemple I.3 Dichotomie

Pour déterminer la solution à ε près de $f(x) = 0$, on peut procéder ainsi :

```
def dichotomie(f, a, b, eps):
  c = (a+b) /2
  if b-a < eps :
    return c
  if f(c) * f(a) > 0 :
    return dichotomie(f, c, b, eps)
  else:
```

```
    return dichotomie(f, a, c, eps)
```

C'est l'algorithme naturel : si l'intervalle est suffisamment petit, on revoie la valeur centrale, sinon on recommence sur la partie sur laquelle f s'annule. ■

■ **Exemple I.4 Palindromes** Pour déterminer si un mot est un palindrome, on peut utiliser le définition suivante :

- tout mot de longueur 0 ou 1 est un palindrome,
- un mot de longueur $n \geqslant 2$ est un palindrome si sa première et dernière lettre sont les mêmes et si le mot obtenu en considérant les lettres 1 à $n-1$ est un palindrome.

Cela donne :

```
def estPalindrome(mot) :
  if len(mot)<= 1 :
    return True
  if mot[0] != mot[-1] :
    return False
  return estPalindrome( mot[1:n-1] )
```

■

■ **Exemple I.5 Recherche d'un élément dans une liste**

Pour savoir si une liste contient un élément nul, on peut utiliser un principe similaire :

```
def contient0(liste) :
  if len(liste) == 0 :
    return False
  elif liste[0] == 0
    return True
  else:
    return contient0(liste[1:])
```

■

R Une version ultra minimaliste de cet algorithme est donc :

```
def contient0(liste) :
  return (len(liste)>0
     and (liste[0] == 0 or contient0(liste[1:])) )
```

■ **Exemple I.6 Algorithme d'Euclide récursif**

Pour calculer le PGCD de (a,b), on utilise la division euclidienne $a = bq + r$ et on sait que le pgcd de (a,b) est celui de (b,r). On est donc ramené au calcul du pgcd de (b,r), jusqu'à ce que le reste soit nul, auquel cas, on renvoie le dernier reste. Cela donne :

```
def euclide(a,b) :
  if b == 0 :
    return a
  else :
    return euclide(b, a%b)
```

■

■ **Exemple I.7 Recherche dichotomique dans une liste triée**

On a une liste triée par ordre croissant, on souhaite savoir si l'élément x est dans la liste. On peut faire exactement comme pour la dichotomie :

```
def recherche(L, x):
    """
    entrée: L = list
              = liste triée dans l'ordre croissant
              x = elmt de L
              = on cherche à savoir si (x in L)
    sortie: True / False selon les cas
    """
    if x < L[0] or  x > L[-1] or len(L) == 0 :
        return False # pas besoin de chercher
    c = len(L) // 2 # indice du milieu
    if L[c] == x:
        return True
    elif L[c]<x :
        return recherche(L[c+1:], x)
    else :
        return recherche(L[:c], x)
```

Si on veut en plus l'indice, on peut utiliser une fonction récursive incluse dans une plus grande fonction (on dit qu'elle est encapsulée).

```
def recherche(L, x):
    """
    entrée: L = list
              = liste triée dans l'ordre croissant
              x = elmt de L
              = on cherche à savoir si (x in L)
    sortie: -1 ou l'indice i tel que L[i] = x
    """
    if x < L[0] or  x > L[-1]:
        return -1 # pas besoin de chercher


    # fct incluse dans une autre:
    def rechercheIter(L, a, b, x):
        """
        entrée (en plus de L et x):
        a, b = deux int
             = avec L[a] <= L[x] <= L[b-1]
             = on cherche si (x in L) dans la partie
                           [| a, b-1|]
        sortie: -1 ou l'indice i tel que L[i] = x
        """
        if a>b-1 :
            # plus d'élément dans la partie considérée
            return -1
```

```
        c = (a+b) // 2 # indice du milieu
        if L[c] == x:
            return c
        elif L[c]<x :
            return rechercheIter(L, c+1, b, x)
        else :
            return rechercheIter(L, a, c, x)
            #NB: c exclu

    return rechercheIter(L,0, len(L), x)
```

■

II Étude théorique d'une fonction récursive

II.1 Terminaison

La terminaison d'un algorithme récursif n'est pas claire : par exemple, la fonction `fac` appelée sur un entier négatif ne finira pas.

> Pour montrer la terminaison d'une fonction récursive, on peut trouver un entier naturel qui décroît strictement à chaque appel de la fonction. Le plus souvent c'est l'un des arguments de la fonction récursive.

■ **Exemple II.1** Pour la fonction factorielle, on constate qu'à chaque appel de la fonction, la valeur de l'argument n diminue de 1. On est donc assuré que la fonction termine au bout de n appels à la fonction. ■

II.2 Correction

> Pour montrer la correction d'un algorithme itératif, on procède généralement par récurrence avec une propriété du type : $P(n)$: la fonction ... appelée sur l'entier n termine et renvoie la valeur correcte.
>
> On prouve alors la correction et la terminaison en même temps.

■ **Exemple II.2** Pour la fonction factorielle, on utilise :

$P(n)$: la fonction `fac` appelée sur l'entier n termine et renvoie $n!$.

L'initialisation pour $n = 0$ est évidente. Considérons $n \geqslant 0$ fixé tel quel $P(n)$ est vrai, la fonction `fac` appelée sur $n+1$ consiste à calculer l'expression `n*fac(n-1)` qui se termine d'après l'hypothèse de récurrence et donne : $n \times (n-1)!$ c'est-à-dire $n!$. ■

II.3 Complexité

> Généralement, on détermine la complexité $C(n)$ du calcul de $f(n)$ par une formule de récurrence.
>
> La complexité en mémoire d'une fonction récursive est liée au nombre d'appels $A(n)$ de la fonction, qui se calcule aussi par récurrence.

Prenons par exemple, le programme vu dans la première section :

```
def u(n) :
  if n == 0
    return 2
  return (1/3) * ( u(n-1) + u(n-1)**2 )
```

On note $C(n)$ le nombre de calculs nécessaires pour l'expression `u(n)`, et $A(n)$ le nombre d'appels. On voit alors :

$$\begin{cases} C(0) = 0 \\ \forall n \in \mathbb{N}^*,\ C(n) = 2C(n-1) + 3 \end{cases} \qquad \begin{cases} A(0) = 0 \\ \forall n \in \mathbb{N}^*,\ A(n) = 2A(n-1) + 2 \end{cases}$$

On a alors facilement

$$\forall n \in \mathbb{N},\ C(n) = 3(2^n - 1) \text{ ainsi } C(n) = O(2^n) \quad A_n = 2^{n+1} - 2 \text{ ainsi } A_n = O(2^n).$$

R En fait, la complexité mémoire est inférieure, car le processeur calcule d'abords la valeur du premier $u(n-1)$ puis libère cette mémoire pour calculer la valeur du deuxième $u(n-1)$.

Reprenons cette exemple, de manière plus efficace :

```
def u(n) :
  if n == 0
    return 2
  x = u(n-1)
  return (1/3) * ( x + x**2 )
```

On note toujours $C(n)$ le nombre de calculs nécessaires pour l'expression `u(n)`, et $A(n)$ le nombre d'appels. On a alors :

$$\begin{cases} C(0) = 0 \\ \forall n \in \mathbb{N}^*,\ C(n) = C(n-1) + 3 \end{cases} \qquad \begin{cases} A(0) = 0 \\ \forall n \in \mathbb{N}^*,\ A(n) = A(n-1) + 1 \end{cases}$$

Et donc on a des valeurs largement inférieures au précédentes :

$$\forall n \in \mathbb{N},\ C(n) = 3n \text{ ainsi } C(n) = O(n) \qquad A_n = n \text{ ainsi } A_n = n.$$

À chaque appel, il y a deux variables crées (x et n), donc la complexité en mémoire est de l'ordre de $2n$.

C'est souvent le cas lorsqu'on utilise un algorithme récursif : la manière dont on le programme modifie la complexité.

Lorsque la récurrence est du type : $C(n)$ s'exprime en fonction de $C\left(\frac{n}{2}\right)$, alors on peut toujours supposer que n est une puissance de 2 et s'écrit $n = 2^k$.

II.4 Amélioration d'un algorithme récursif

Considérons l'exemple de **La suite de Fibonnacci**

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

L'algorithme récursif de calcul est alors le suivant :

```
def fibbo(n):
  if n == 0 :
    return 0
  elif n == 1:
    return 1
  else :
    return fibbo(n-1) + fibbo(n-2)
```

Pour montrer que cet algorithme se termine et est correct, on utilise la propriété :

$$\mathscr{P}(n) : \text{"}\texttt{fibbo(n)} \text{ se termine et est égal à } F_n$$

que l'on démontre par récurrence double.

Cet algorithme est catastrophique en terme de temps de calcul. Si $C(n)$ est la nombre de calcul nécessaire pour calculer `fibbo(n)`, on a :

$$\forall n \in \mathbb{N}, C(n+2) = C(n) + C(n+1) + 1.$$

On écrit alors :

$$(C(n+2)+1) = (C(n)+1) + (C(n+1)+1)$$

si bien que la suite $u_n = (C(n)+1)$ vérifie alors la relation : $u_{n+2} = u_{n+1} + u_n$ et est donc récurrente linéaire d'ordre 2.

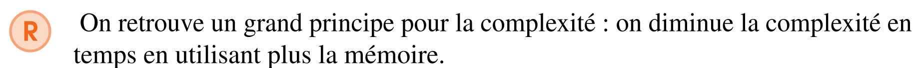
On considère donc le polynôme $X^2 - X - 1$ et on note $\varphi = \frac{1+\sqrt{5}}{2}$ la plus grande racine du polynôme, l'autre est $-\frac{1}{\varphi}$. On sait alors que :

$$\exists (\alpha, \beta) \in \mathbb{R}^2,\ u_n = \alpha \varphi^n + (-1)^n \beta \frac{1}{\varphi^n}$$

et donc que $u_n \underset{n\infty}{\sim} \alpha \varphi^n$, et comme $C(n) = u_n + 1$, cela donne que la suite $C(n)$ est alors $\underset{n\infty}{O}(\varphi^n)$ où φ est le nombre d'or.

C'est clairement une trop grande complexité.

On voit bien le problème : pour calculer F_n, l'algorithme fait le calcul de F_{n-1}, puis de F_{n-2}. Lors du deuxième calcul de F_{n-2}, il ne se sert pas des valeurs déjà calculées il repart à 0. Pour améliorer cet algorithme une idée naturelle est de stocker les valeurs de F_n calculés dans un tableau pour éviter de les recalculer.



R On retrouve un grand principe pour la complexité : on diminue la complexité en temps en utilisant plus la mémoire.

Le programme devient alors :

```
NMAX = 100
nbrAppel = 0

# tabFibbo va contenir les valeurs connues de
# fibbo (n)
tabFibbo = [-1] * NMAX
tabFibbo[0]=0
tabFibbo[1]=1
```

```
def fibbo(n):
    """
    n <= NMAX !
    """
#   global tabFibbo
    global nbrAppel
    nbrAppel += 1
    if tabFibbo[n] == -1 : # fibbo(n) est inconnu
        tabFibbo[n] = fibbo(n-1) + fibbo(n-2)
    return tabFibbo[n]

print(fibbo(99))
print(nbrAppel)
```

Plusieurs technique existent pour utiliser le tableau :

- utiliser une variable globale comme ci-dessus,
- modifier la liste à l'aide d'une méthode (on peut ainsi modifier la liste dans l'espace d'exécution supérieur), ce qui revient à utiliser une variable globale. Dans le programme ci-dessus, l'instruction `global tabFibbo` est ainsi facultative.
- Utiliser une fonction `fibboRec` à l'intérieur de la fonction `fibbo`. C'est la fonction `fibbo` qui initialise le tableau, celle-ci est utilisé dans l'espace d'exécution inférieur de la fonction `fibboRec`.

Voici un exemple de solution avec la dernière méthode :

```
def fibbo(n):


    tabFibbo = [-1] * (n+1)
    tabFibbo[0]=0
    tabFibbo[1]=1

    def fibboRec(n):
        if tabFibbo[n] == -1 : # fibbo(n) est inconnu
            tabFibbo[n] = fibboRec(n-1) + fibboRec(n-2)
        return tabFibbo[n]
    return fibboRec(n)

print(fibbo(99))
```

III Exemple : algorithme itératif du triangle de Pascal

On cherche à générer les parties de $[\![1,n]\!]$ qui contiennent k éléments, ensemble d'ensembles noté $\mathscr{P}_k([\![1,n]\!])$ qui sera représenté sous la forme d'une liste de liste.

Toues les éléments X de $\mathscr{P}_k([\![1,n]\!])$, qui sont donc des parties de $[\![1,n]\!]$ à k éléments, vérifient une et une seule des conditions suivantes :

- Soit $n \notin X$ et donc essentiellement, X est une partie de $[\![1,n-1]\!]$ à k éléments,
- Soit $n \in X$ et donc si on enlève n, l'ensemble $X \setminus \{n\}$ est une partie de $[\![1,n-1]\!]$ à $k-1$ éléments. Vu d'un autre côté, X est obtenu en considérant une partie à $k-1$ éléments de $[\![1,n-1]\!]$ à laquelle on a ajouté n.

C'est l'interprétation combinatoire de la propriété de Pascal sur les coefficient binomiaux : $\mathscr{P}_k(\llbracket 1,n \rrbracket)$ est l'union disjointe de

- $\mathscr{P}_k(\llbracket 1,n-1 \rrbracket)$ l'ensemble des parties de $\llbracket 1,n-1 \rrbracket$ à k éléments,
- un autre ensemble noté B, qui est l'ensemble des parties de $\llbracket 1,n-1 \rrbracket$ à $k-1$ éléments auxquelles on a jouté n. On a de manière évidente : B est en bijection avec $\mathscr{P}_{k-1}(\llbracket 1,n-1 \rrbracket)$.

Ce qui donne la formule bien connue :

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}.$$

Cet démonstration mathématique se transforme naturellement en algorithme car de même que pour le calcul des coefficients $\binom{n}{k}$, on connaît les ensembles à 0 éléments et à n éléments. Cela donne :

```
def genereParties(k , n):
    """
    entrée: entiers k entiers n
    sortie: liste de liste
            = les combinaisons de k éléments de 1 n .
    """
    if k==0 :
        # seul ensemble à 0 élément est l'ensemble vide
        return [[]] #
    elif k==n :
        # seul ensemble à n élément de [|1,n|] est [|1,n|]
        return [ list(range(1,n+1)) ]
    else :
        #calcul des listes de k éléments de [|1, n-1|]
        L1 = genereParties(k , n-1)
        #calcul des listes de k-1 éléments de [|1, n-1|]
        #auquel on ajoute n
        L2 = genereParties(k-1 , n-1)
        for l in L2 :
            l.append(n)
        return L1 + L2
```

R L'utilisation d'objets mutables et de références permet d'écrire :

```
for l in L2 :
  l.append(n)
```

pour ajouter `n` à la fin de toutes les liste de `L2`.

Autre détail technique : la fonction `range` renvoie un itérateur, pour obtenir une liste, on convertit avec `list(range(1,n+1))`

Pour prouver la terminaison, on peut utiliser la propriété suivante :

$\mathscr{P}(n)$: "$\forall k \in \llbracket 0,n \rrbracket$, `genereParties(k,n)` se termine et renvoie le bon résultat.

C'est alors vrai pour $n = 0$ (il n'y a alors que $k = 0$ comme valeur possible).

Soit n fixé, tel que $\mathscr{P}(n-1)$ est vrai, et considérons $k \in \llbracket 0,n \rrbracket$.

Si $k = 0$ ou $k = n$, alors `genereParties(k,n)` se termine et renvoie la bonne valeur. On suppose donc $k \llbracket 1,n-1 \rrbracket$. Par hypothèse de récurrence, les instructions :

```
L1 = genereParties(k , n-1)
L2 = genereParties(k-1 , n-1)
```

se terminent et donnent le bon résultat (en effet k et $k-1$ sont dans $[\![0,n-1]\!]$). Ainsi, `genereParties(k,n)` se termine. L'étude mathématique assure alors que `genereParties(k,n)` renvoie la bonne valeur.

Pour prouver que cet algorithme se termine, on peut constater que la valeur $\max(k,n)$ est décroissante strictement à chaque appel de la fonction.

Comme on peut le voir cet algorithme est élégant mais loin d'être optimal pour la compléxité.

Exercice 1 Modifier le programme, pour garder en mémoire les valeurs connues de `genereParties(k , n)`.

IV Exemple Exponentiation rapide

On souhaite calcule x^n avec x un réel et n un entier. On utilise alors la méthode suivante :

$$x^n = \begin{cases} \left(x^{\frac{n}{2}}\right)^2 & \text{si } n \text{ est pair} \\ x \times \left(x^{\frac{n}{2}}\right)^2 & \text{si } n \text{ est impair} \end{cases}$$

Cela donne ainsi l'algorithme :

```
def exponentiation(x,n) :
  if n == 0 :
    return 1
  r = exponentiation(x , n//2)
  if n%2 == 0 :
    return r*r
  else :
    return x*r*r
```

Pour prouver la correction et la terminaison de cet algorithme, on considère $x \in \mathbb{R}$, et on utilise la proposition suivante :

$P(n)$: `exponentiation(x,n)` se termine et donne x^n.

On raisonne alors par récurrence forte :

- $P(0)$ est vrai.
- Soit n fixé, tel que $\forall k \in [\![0,n]\!]$, $P(k)$ est vrai. Si n est pair, on écrit $n = 2k$, et en utilisant l'hypothèse $P(k)$, on obtient que l'expression `r = exponentiation(x , n//2)` se termine et donne x^k. D'où l'expression `r*r` se termine et donne bien x^{2k} ie x^n. Si n est impair, on écrit $n = 2k+1$, et on a toujours `r = exponentiation(x , n//2)` se termine et donne x^k. D'où l'expression `x*r*r` se termine et donne bien x^{2k+1} ie x^n.
- D'où le résultat.

Si on note $C(n)$ le nombre de calcul nécessaire pour obtenir x^n, on a :

$$C(0) = 0 \qquad \text{et} \qquad \forall n \in \mathbb{N}^*,\ C(n) \leqslant C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 2$$

On suppose que n est une puissance de 2 : $n = 2^k$, et on note $u_k = C(2^k)$, on a alors : $u_k = u_{k-1} + 2$, et donc $u_k \underset{k\infty}{=O} (k)$, ce qui donne $C(n) \underset{n\infty}{=O} (\log_2(n))$.

NB : on admet que l'on peut étendre ce résultat valable dans le cas où n est une puissance de 2 à un n quelconque dans $\mathbb{N}$ (avec un « grand o »).

Récursivité

Pelletier Sylvain PSI, LMSC

Exercice 1 Écrire une fonction récursive qui calcule la somme des n premiers entiers. Prouver la correction et déterminer le nombre d'opérations et d'appels nécessaires.

Exercice 2 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.

Exercice 3 Liste miroir

Écrire une fonction récursive qui prends en entrée une liste `L` et qui construit la liste comportant les mêmes éléments que `L` mais dans l'ordre inverse. Bien entendu, on s'interdit d'utiliser la fonction `reverse`.

Exercice 4 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.

On a retrouvé la formule : $n! = n \times (n-1)!$.

Exercice 5 Soit n un entier naturel non nul, une **combinaison** de l'entier n est une liste $(n_1, \ldots, n_p)$ 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 $\Big\{(1)\Big\}$, ce qui s'écrit en Python `[[1]]`.

Pour $n = 2$, les combinaisons sont (2) et $(1,1)$, ce qui s'écrit : $\Big\{(2),(1,1)\Big\}$, 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]]
```

Algorithme de tri par sélection
Tri par insertion
Tri fusion
Tri rapide
Comparatif des algorithmes de tris
Recherche de la médiane

4 — Algorithmes de tris

Dans ce chapitre, on étudie un problème important en informatique : trier les éléments d'une liste L.

On va étudier et comparer plusieurs algorithmes permettant de réaliser cette tâche.

L'étude de ces algorithmes va permettre de revoir plusieurs grandes idées de l'algorithmique vues en première année (preuve de la terminaison, de la correction et calcul de la complexité).

Mais le tri n'est pas qu'un problème académique : en pratique, un ordinateur passe en moyenne un tiers de son temps à trier.

On va noter n la taille de la liste, on a donc : $L = [L_0, \dots, L_{n-1}]$.

On mesure **la complexité en nombre de comparaisons**, c'est-à-dire le nombre de test du type `L[i] < L[j]` que l'on fait. On considère ainsi que l'échange de deux éléments dans la liste et que l'accès à l'élément i est immédiat,

Au niveau de la complexité en mémoire, on parle de **tri sur place**, lorsque la liste L est triée sans utiliser d'espace supplémentaire : l'algorithme ne requiert qu'un espace en mémoire constant pour quelques variables.

Pour chaque algorithme, il faut savoir les appliquer sur un exemple, savoir les programmer et comprendre leur étude théorique (complexité et preuve de correction).

R Par convention, on classe la liste par ordre croissant.

Dans le cours on suppose que l'on trie des réels. En pratique, on trie des éléments d'un type quelconque sur lequel est défini un ordre total.

Le choix de ne compter que les comparaisons est contestable : il correspond au cas de données faciles d'accès (pour lecture et écriture) en mémoire. Si ce n'est pas le cas la performance des algorithmes est différente.

✯ Méthode sort

Pour une liste L, on dispose de la méthode `sort` qui permet de trier la liste (sur place) et de la fonction `sorted` qui crée une copie triée de la liste.

On peut choisir de trier par ordre décroissant, ou d'indiquer une méthode de tri :

La syntaxe est : `L.sort(key=None, reverse=False)`. L'argument optionnel `key` est une fonction qui prends un élément de la liste x et qui est appliquée sur chaque élément avant de faire des comparaison. C'est ainsi la fonction qui indique le critère de tri. L'argument optionnel `reverse` indique si il faut trier par ordre croissant ou décroissant.

La fonction `sorted` a la même syntaxe :
`sorted(iterable, key=None, reverse=False)`.

■ **Exemple .1** Pour trier une liste de 3-uplets (i, j, k) en fonction de la troisième composante par ordre décroissant, on fait :

```
L.sort(key = lambda x: x[2], reverse=True)
```

■

Algorithme de tri par sélection

✯ Principe de l'algorithme

Le tri par sélection consiste à :
- chercher l'élément minimum de la liste L, noté L_j
- le déplacer en première place, en échangeant l'élément 0 et l'élément j,
- recommencer sur les éléments $[L_1, \ldots, L_n]$

On rappelle que pour chercher un minimum :
- on considère que le minimum est atteint à la première position,
- on conserve sa valeur (ou sa position) dans une variable.
- Puis on balaie les éléments suivants,
- et on met à jour le minimum selon la valeur lue.

Ici, on va utiliser une variable $i \in [\![0, n-2]\!]$, en considérant qu'à l'itération i, les éléments de $[\![0, i-1]\!]$ sont triés. On cherche donc le minimum parmi les éléments de $[\![i, n-1]\!]$.

✯ Réalisation

Voici un exemple d'algorithme de tri par sélection :

```
def tri_selection(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]
```

⋆ Preuve de la correction

Proposition I.1 Un invariant de boucle est :

$\mathscr{P}(i)$: À la fin de la boucle indéxée par i (ie ligne 11), on a :

$$L_0 \leqslant L_1 \leqslant \cdots \leqslant L_i \leqslant \min\left(L_{i+1}, \ldots, L_{n-1}\right).$$

Autrement dit, les i premiers éléments sont bien placés et donc à la dernière itération tous les éléments sont bien placés.

Ainsi, lorsque $i = n-2$, la liste est triée.

Démonstration. C'est une démonstration par récurrence sur i. En notant :

$$\mathscr{P}(i): \quad L_0 \leqslant L_1 \leqslant \cdots \leqslant L_i \leqslant \min\left(L_{i+1}, \ldots, L_{n-1}\right)$$

$\mathscr{P}(0)$ est dû à la preuve de la recherche du minimum.

Si on considère un i fixé, tel que $\mathscr{P}(i)$ est vrai, alors l'algorithme de recherche de minimum assure que $\mathscr{P}(i+1)$ est vrai au passage suivant.

■

⋆ Étude de la complexité

Proposition I.2 La complexité en nombre de comparaison de l'algorithme de tri est : $C(n) = \dfrac{n(n-1)}{2} \underset{n \to +\infty}{\sim} \dfrac{n^2}{2}$.

Démonstration. Pour l'étape $i \in [\![0, n-1]\!]$, il y a $n-i-1$ comparaisons.

Ainsi,

$$C(n) = \sum_{i=0}^{n-1}(n-i-1) = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}.$$

■

⋆ Application sur un exemple

Sur la liste $L = [13, 1, 5, 6, 2, 6, 7, 4, 3, 23, 14, 23, 34]$, voici les étape :

```
L: [14, 9, 7, 13, 10, 1, 8, 3, 11, 12, 6, 4, 5, 2]
étape 0 L : [1, 9, 7, 13, 10, 14, 8, 3, 11, 12, 6, 4, 5, 2]
étape 1 L : [1, 2, 7, 13, 10, 14, 8, 3, 11, 12, 6, 4, 5, 9]
étape 2 L : [1, 2, 3, 13, 10, 14, 8, 7, 11, 12, 6, 4, 5, 9]
étape 3 L : [1, 2, 3, 4, 10, 14, 8, 7, 11, 12, 6, 13, 5, 9]
étape 4 L : [1, 2, 3, 4, 5, 14, 8, 7, 11, 12, 6, 13, 10, 9]
étape 5 L : [1, 2, 3, 4, 5, 6, 8, 7, 11, 12, 14, 13, 10, 9]
étape 6 L : [1, 2, 3, 4, 5, 6, 7, 8, 11, 12, 14, 13, 10, 9]
étape 7 L : [1, 2, 3, 4, 5, 6, 7, 8, 11, 12, 14, 13, 10, 9]
étape 8 L : [1, 2, 3, 4, 5, 6, 7, 8, 9, 12, 14, 13, 10, 11]
étape 9 L : [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 14, 13, 12, 11]
étape 10 L : [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 12, 14]
étape 11 L : [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
étape 12 L : [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

II Tri par insertion

✯ Principe de l'algorithme

Le tri par insertion consiste à exploiter le fait qu'il est facile de placer un élément dans une liste déjà triée.

On va donc trier progressivement la liste : le début de la liste est déjà trié et on insère un nouvel élément à la bonne place parmi les premiers.

On procède alors ainsi :

- le premier élément est trié (puisqu'il est seul).
- On insère le deuxième élément avant ou après le premier, les deux premiers éléments sons alors triés.
- On insère les troisième élément à la bonne place par rapport aux deux premiers, les trois premiers éléments sont alors triés,
- on insère le quatrième à la bonne place par rapport au trois premiers éléments, etc.

Ainsi, à l'étape $i \in [\![1, n-1]\!]$, les $i-1$ premiers éléments sont triés entre eux et on insère le i-ème éléments à la bonne place par rapport aux $i-1$ premiers.

✯ Réalisation

Pour plus de facilité, on utilise deux fonctions :

- la première `deplace` prend en entrée la liste L et une position i (dans la liste L) : on suppose que les éléments $L[0], \ldots, L[i-1]$ sont déjà triés et on déplace l'élément $L[i]$ pour qu'il soit à la bonne place dans cette partie (éventuellement en première ou dernière place).
  Pour cela :
  - on suppose on repère sa position par la variable j, on peut dire que l'on utilise une « tête de lecture » qui lit la liste sur en j et $j-1$. Au début $j = i$.
  - tant que la tête de lecture est toujours dans la liste et que `L[j] < L[j-1]` (ie la tête de lecture lit deux éléments dans le mauvais ordre), on échange les éléments place $j-1$ et j (ie on déplace l'élément vers la gauche), on déplace aussi la tête de lecture de telle sorte qu'elle est toujours placée pour comparer l'élément que l'on déplace et l'élément à sa gauche.
- la deuxième fonction trie la liste en insérant chaque élément. Il s'agit d'un simple apel dans une boucle `for` à la fonction `deplace`.

Voici un exemple de code :

```
def deplace(L, i):
    """
    entrée:
    L = list
      = liste dont les éléments L[0] ... L [i-1]
        sont déjà triés
    i = int
      = un indice dans la liste.
    sortie: rien, la liste L est modifiée
    """
    j = i
    while j >0 and L[j] < L[j-1]:
      L[j-1], L[j] = L[j], L[j-1]
```

```
            j -= 1


def tri(L) :
    """
    entrée: L = liste
              = liste à trier
    sortie: L triée
              la liste L est modifiée sur place
    """
    n = len(L)
    for i in range(1,n) :
        deplace(L,i)
    return L
```

✯ Preuve de la terminaison

La terminaison de `deplace` est évidente : la valeur de j diminue à chaque itération. Il y a au plus i itérations. La terminaison de `tri` est évidente aussi : c'est une boucle `for`.

✯ Preuve de la correction

Pour prouver la correction de cet algorithme, on peut commencer par prouver la correction de déplace :

Proposition II.1 La fonction `deplace` est correcte.
Un invariant de boucle au début de la boucle `while` est :

$$\texttt{L}[\texttt{j}] \text{ est l'élément que l'on déplace (l'ancien élément } \texttt{L}[\texttt{i}])$$
$$L[j] \leqslant L[j+1] \leqslant \cdots \leqslant L[i] \text{ et } L[0] \leqslant L[1] \leqslant \cdots \leqslant L[j-1]$$

Démonstration. C'est vrai avant de rentrer dans la boucle pour la valeur $j = i$.

Supposons que ce soit vrai pour une valeur fixée de j, on sait alors que : `L[j]` est l'élément que l'on déplace, et que :

$$L[j] \leqslant L[j+1] \leqslant \cdots \leqslant L[i] \text{ et } L[0] \leqslant L[1] \leqslant \cdots \leqslant L[j-1]$$

Comme on n'est pas sorti de la boucle, on sait aussi que $L[j-1] < L[j]$.

On échange alors $L[j-1]$ et $L[j]$ si bien que après échange, on a :

$$L[j-1] \leqslant L[j] \leqslant L[j+1] \leqslant \cdots \leqslant L[i]$$

et toujours :

$$L[0] \leqslant L[1] \leqslant \cdots \leqslant L[j-2]$$

D'où l'hérédité.

On en déduit que c'est un invariant de boucle et donc que la propriétée est vraie en sortie. Si $j = 1$ en sortie, on a alors :

$$L[j-1] \leqslant L[j] \leqslant L[j+1] \leqslant \cdots \leqslant L[i] \text{ s'écrit : } L[0] \leqslant L[1] \leqslant L[j+1] \leqslant \cdots \leqslant L[i]$$

et si $L[j-1] \leqslant L[j]$, on a :

$$\begin{cases} L[j] \leqslant L[j+1] \leqslant \cdots \leqslant L[i] \\ L[0] \leqslant L[1] \leqslant \cdots \leqslant L[j-1] \\ L[j-1] \leqslant L[j] \end{cases} \quad \text{donc } L[0] \leqslant L[1] \leqslant L[j+1] \leqslant \cdots \leqslant L[i]$$

La fonction `deplace` est donc correcte. ■

On en déduit la correction de la fonction `tri`

Proposition II.2 La fonction `tri` est correcte. Un invariant de boucle est : $\mathscr{P}(i)$: après utilisation de la fonction `deplace`

$$L_0 \leqslant L_1 \leqslant \cdots \leqslant L_i$$

Autrement dit les i premiers éléments de la liste sont triés.

Démonstration. Par récurrence sur i c'est évident, puisque la fonction `deplace` place l'élément `L[i]` au bon endroit. ■

✯ Complexité

On étudie la complexité dans le meilleur et dans le pire des cas.

Proposition II.3 Dans le pire des cas, la complexité de l'algorithme de tri par insertion est :

$$C(n) = \frac{n(n-1)}{2} \underset{n \to +\infty}{\sim} \frac{n^2}{2}$$

Dans le meilleur des cas, la complexité de l'algorithme de tri par insertion est :

$$C(n) = n - 1 \underset{n \to +\infty}{\sim} n$$

Démonstration. Dans le pire des cas, à l'itération i, la variable j va varier de i à 1, il va donc y avoir i test.

Ce qui donne :

$$C(n) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}.$$

Dans le meilleur des cas, à l'itération i, la variable j ne va pas varier (un seul test est effectué). Ce qui donne :

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1.$$

■

R Le pire des cas est obtenu quand la liste est triée par ordre décroissant. Le meilleur des cas est obtenu quand la liste est déjà triée par ordre croissant.

★ Tri par insertion dichotomique

En fait le tri par insertion peut être grandement améliorer en cherchant la position j de l'élément i dans les $i-1$ premiers éléments déjà triés en utilisant la dichotomie.

Exercice 1 Écrire la fonction `place` correspondante. Calculer la complexité.

★ Application sur un exemple

```
L: [1, 8, 2, 6, 12, 3, 9, 7, 13, 10, 5, 14, 11, 4]
étape  1 L: [1, 8, 2, 6, 12, 3, 9, 7, 13, 10, 5, 14, 11, 4]
étape  2 L: [1, 2, 8, 6, 12, 3, 9, 7, 13, 10, 5, 14, 11, 4]
étape  3 L: [1, 2, 6, 8, 12, 3, 9, 7, 13, 10, 5, 14, 11, 4]
étape  4 L: [1, 2, 6, 8, 12, 3, 9, 7, 13, 10, 5, 14, 11, 4]
étape  5 L: [1, 2, 3, 6, 8, 12, 9, 7, 13, 10, 5, 14, 11, 4]
étape  6 L: [1, 2, 3, 6, 8, 9, 12, 7, 13, 10, 5, 14, 11, 4]
étape  7 L: [1, 2, 3, 6, 7, 8, 9, 12, 13, 10, 5, 14, 11, 4]
étape  8 L: [1, 2, 3, 6, 7, 8, 9, 12, 13, 10, 5, 14, 11, 4]
étape  9 L: [1, 2, 3, 6, 7, 8, 9, 10, 12, 13, 5, 14, 11, 4]
étape  10 L: [1, 2, 3, 5, 6, 7, 8, 9, 10, 12, 13, 14, 11, 4]
étape  11 L: [1, 2, 3, 5, 6, 7, 8, 9, 10, 12, 13, 14, 11, 4]
étape  12 L: [1, 2, 3, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 4]
étape  13 L: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

III Tri fusion

★ Principe de l'algorithme

Le tri fusion consiste à appliquer une méthode itérative pour trier : on découpe la liste « au milieu » et on trie chaque partie.

Ensuite, on fusionne les deux listes obtenues en un nombre linéaire de comparaison.

Autrement dit, étant donné une liste de taille n, on applique le tri sur les sous-listes `L[: n//2]` et `L[n//2:]`. On renvoie les deux listes fusionnées. Si la liste est vide ou ne contient qu'un élément on la retourne inchangée.

On a encore deux fonctions :

- La fonction `fusionne`, qui fusionne deux listes triées,
- et la fonction `tri`, qui trie la liste

Pour fusionner deux listes `L1` et `L2`, on construit une liste triée `Lt` en utilisant les deux liste `L1` et `L2` comme des piles :

- on regarde les premiers éléments de chacune des piles (attention la convention est différente qu'au chapitre pile : le haut de la pile est le premier élément de la liste),
- on dépile le plus petit des deux que l'on ajoute à la liste triée `Lt` (à la fin ici)
- jusqu'à ce qu'un des deux piles soit vide,
- on ajoute enfin la pile non vide à `Lt`.

La fonction `tri` est une simple fonction récursive :

- Si la liste est vide ou ne contient qu'un élément, on la renvoie,

- sinon, on calcule le position du point milieu, on relance la fonction sur chaque partie et on renvoie la fusion des deux parties triées.

✭ Réalisation

Voici un exemple de code :

```
def fusion(L1, L2):
    """
    entrée: L1, L2 = list
            = liste déjà triée
    sortie: Lt = list
            = liste triée obtenue en fusionnant
              les deux listes L1 et L2
    """
    Lt= []
    while len(L1)>0 and len(L2)> 0:
        if L1[0] < L2[0] :
            # on prend le premier élément de L1
            elmt = L1.pop(0)
        else:
            # idem avec L2
            elmt = L2.pop(0)
        # on l'ajoute à Lt
        Lt.append(elmt)

    # on ajoute les derniers éléments
    if len(L1)>0 :
        Lt.extend(L1)
    else :
        Lt.extend(L2)
    return Lt

def tri(L) :
    """
    entrée: L = list
              = liste à trier
    sortie: liste triée
    """
    n = len(L)
    if n<=1 :
        return L
    m = n//2 # point milieu
    debut = tri(L[:m])
    fin = tri( L[m:])
    return fusion( debut , fin )
```

✭ Preuve de la terminaison

Proposition III.1 La fonction `fusion` appliquée à deux liste `L1` et `L2` classées par ordre croissant se termine.

La fonction `tri` appliquée à une liste `L` se termine.

Démonstration. Pour la fonction `fusion`, on enlève un terme à `L1` ou à `L2` à chaque

passage dans la boucle `while`. La boucle `while` se termine donc.

Pour la fonction `tri`, on utilise la récurrence :

$\mathscr{P}(n)$ appliquée sur une liste `L` de longueur inférieure ou égale à n, la fonction `tri` se termine.

C'est vrai pour $n = 1$ ou $n = 2$.

Si $n \geqslant 2$ vérifie que $\mathscr{P}(k)$ est vrai pour tout les rangs $k < n$. Alors c'est vrai pour les rangs $\lfloor \frac{n}{2} \rfloor$ et $\lfloor \frac{n}{2} \rfloor + 1$. On en déduit $\mathscr{P}(n)$.

Par récurrence forte, c'est donc vrai pour toute valeur de n. ■

⋆ Preuve de la correction

Proposition III.2 La fonction `fusion` appliquée à deux liste `L1` et `L2` classées par ordre croissant renvoie bien la bonne valeur.

La fonction `tri` appliquée à une liste `L` renvoie bien la bonne valeur.

Démonstration. Pour la fonction `fusion` on a un invariant de boucle dans la boucle `while` à la fin de l'itération i :

$$\begin{aligned} & Lt[0] < Lt[1] < \cdots < Lt[i] \\ & \text{et } Lt[i] \leqslant L1[0] \text{ et } Lt[i] \leqslant L2[0] \\ & \text{et } len(L1) + len(L2) \text{ a diminué de } i. \end{aligned}$$

Comme à chaque itération on extrait la plus petite valeur `L1[0]` ou `L2[0]`, et que les listes sont stockées par ordre croissant la propriété est bien récursive.

Pour `tri`, on procède par récurrence :

$\mathscr{P}(n)$ appliquée sur une liste `L` de longueur inférieure ou égale à n, la fonction `tri` renvoie le bon résultat.

C'est vrai pour $n = 1$ ou $n = 2$.

Si $n \geqslant 2$ vérifie que $\mathscr{P}(k)$ est vrai pour tout les rangs $k < n$. Alors c'est vrai pour les rangs $\lfloor \frac{n}{2} \rfloor$ et $\lfloor \frac{n}{2} \rfloor + 1$. On fusionne alors deux listes triées par ordre croissant pour obtenir une liste triée par ordre croissant. On en déduit $\mathscr{P}(n)$. Par récurrence forte, c'est donc vrai pour toute valeur de n.

■

⋆ Complexité

Proposition III.3 On a les compléxité suivantes (en nombre de comparaisons) pour l'algorithme de fusion :

- Dans le pire des cas : $\texttt{len(L1)} + \texttt{len(L2)}$ comparaisons.
- Dans le meilleur des cas : $\min(\texttt{len(L1)}, \texttt{len(L2)})$ comparaisons.

Démonstration. À chaque passage dans la boucle `while`, la taille d'une des listes est diminuée de 1, jusqu'à que l'un des liste soit vide. On fait une comparaison par passage.

Ainsi, le nombre de comparaisons $C(n)$ vérifie :

$$\min(\texttt{len(L1)}, \texttt{len(L2)}) \leqslant C(n) \leqslant \texttt{len(L1)} + \texttt{len(L2)}.$$

■

Le meilleur des cas arrive lorsque on enlève systématiquement des éléments à la même liste, cela corresponds à dire que tous les éléments de `L1` sont inférieurs aux éléments de `L2`.

Le pire des cas arrive lorsque on enlève tous les éléments des deux listes. Cela corresponds au cas où les deux listes sont « équilibrées ».

> **Proposition III.4** La complexité $C(n)$ de l'algorithme de tri fusion vérifie $C(n) = O(n\log_2(n))$, dans le meilleur et dans le pire des cas.

Démonstration. On traite le pire des cas.

La complexité de la fonction `tri` sur une liste de longueur n vérifie :

$$C(n) = 2\times C\left(\frac{n}{2}\right) + n$$

Puisqu'on fusionne deux listes dont la somme des longueurs fait n.

Ainsi, on a :

$$C\left(2^k\right) = 2C(2^{k-1}) + 2^k$$

On écrit rapidement :

$$\begin{aligned}
C\left(2^k\right) &= 2C(2^{k-1}) + 2^k && L_k\\
C\left(2^{k-1}\right) &= 2C(2^{k-2}) + 2^{k-1} && L_{k-1}\\
C\left(2^{k-2}\right) &= 2C(2^{k-3}) + 2^{k-2} && L_{k-2}\\
\vdots &= \vdots + \vdots &&\\
C(2) &= 2C(1) + 2 && L_1\\
C(1) &= 0 && L_0
\end{aligned}$$

En faisant

$$L_k + 2L_{k-1} + 4L_{k-2} + \cdots + 2^k L_0$$

On obtient :

$$C\left(2^k\right) = 2^k + 2^k + \cdots + 2^k = k2^k.$$

Ainsi, si n s'écrit sous la forme : $n = 2^k$, on a :

$$C(n) = n\log_2(n)$$

On souhaite généraliser cette formule à un $n \in \mathbb{N}$.

On note alors $k \in \mathbb{N}$ l'entier tel que $2^k \leqslant n < 2^{k+1}$. On sait alors que $k = \lfloor \log_2(n) \rfloor$

On utilise alors la croissance de la fonction complexité, qui donne :

$$\begin{aligned}
\text{on a : } 2^k &\leqslant n < 2^{k+1}\\
\text{donc } C\left(2^k\right) &\leqslant C(n) < C\left(2^{k+1}\right)\\
\text{ce qui donne : } k2^k &\leqslant C(n) < (k+1)2^{k+1}\\
\text{en remplaçant : } \lfloor \log_2(n) \rfloor n &\leqslant C(n) < (\lfloor \log_2(n) \rfloor + 1)(2n)\\
(\log_2(n) - 1)n &\leqslant C(n) < 2(\log_2(n) + 2)n
\end{aligned}$$

D'où $C(n) = O(n\log_2(n))$.

Dans le meilleur des cas, la complexité de la fonction `tri` sur une liste de longueur n vérifie :

$$C(n) = 2 \times C\left(\frac{n}{2}\right) + \frac{n}{2}$$

Puisqu'on fusionne deux liste de longueur $\frac{n}{2}$. En reprenant le calcul, on obtient de même : $C(n) = O(n\log_2(n))$.

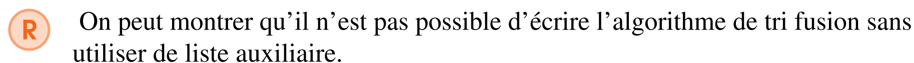
■

R L'écart entre le meilleur et le pire des cas est simplement un facteur 2.

Ici, le calcul de la complexité a été très détaillé. En pratique, on peut se contenter d'indiquer comme résultat de complexité : si n s'écrit sous la forme : $n = 2^k$, alors $C(n) = n\log_2(n)$. Il n'est pas nécessaire de détailler la généralisation à tout $n \in \mathbb{N}$. La technique pour généraliser ici est très classique, mais n'est nécessaire que si elle est explicitement demandée.

Proposition III.5 Le tri par fusion n'est pas un tri sur place : il est nécessaire de stocker une liste auxiliaire de la même taille que `L`.

Démonstration. Dans la fonction `fusion` on a utilisé la liste intermédiaire `Lt`. ■


R On peut montrer qu'il n'est pas possible d'écrire l'algorithme de tri fusion sans utiliser de liste auxiliaire.

⋆ Test sur un exemple

Voici une illustration sur un exemple :

```
L: [1, 11, 6, 5, 12, 13, 8, 14, 9, 7, 2, 3, 10, 4]
on trie: [1, 11, 6, 5, 12, 13, 8, 14, 9, 7, 2, 3, 10, 4]
on trie: [1, 11, 6, 5, 12, 13, 8]
on trie: [1, 11, 6]
on trie: [1]
on trie: [11, 6]
on trie: [11]
on trie: [6]
on fusionnne [11] et [6]
on obtient: [6, 11]
on fusionnne [1] et [6, 11]
on obtient: [1, 6, 11]
on trie: [5, 12, 13, 8]
on trie: [5, 12]
on trie: [5]
on trie: [12]
on fusionnne [5] et [12]
on obtient: [5, 12]
on trie: [13, 8]
on trie: [13]
```

```
on trie: [8]
on fusionnne [13] et [8]
on obtient: [8, 13]
on fusionnne [5, 12] et [8, 13]
on obtient: [5, 8, 12, 13]
on fusionnne [1, 6, 11] et [5, 8, 12, 13]
on obtient: [1, 5, 6, 8, 11, 12, 13]
on trie: [14, 9, 7, 2, 3, 10, 4]
on trie: [14, 9, 7]
on trie: [14]
on trie: [9, 7]
on trie: [9]
on trie: [7]
on fusionnne [9] et [7]
on obtient: [7, 9]
on fusionnne [14] et [7, 9]
on obtient: [7, 9, 14]
on trie: [2, 3, 10, 4]
on trie: [2, 3]
on trie: [2]
on trie: [3]
on fusionnne [2] et [3]
on obtient: [2, 3]
on trie: [10, 4]
on trie: [10]
on trie: [4]
on fusionnne [10] et [4]
on obtient: [4, 10]
on fusionnne [2, 3] et [4, 10]
on obtient: [2, 3, 4, 10]
on fusionnne [7, 9, 14] et [2, 3, 4, 10]
on obtient: [2, 3, 4, 7, 9, 10, 14]
on fusionnne [1, 5, 6, 8, 11, 12, 13] et [2, 3, 4, 7, 9,
 10, 14]
on obtient: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

IV Tri rapide

✯ Principe de l'algorithme

Le tri rapide (*quicksort*) consiste à utiliser un processus récursif :

- choisir un élément x comme **pivot** (généralement le dernier, on peut aussi le choisir au hasard),
- cet élément va permettre de partitionner la liste L en deux parties. On va construire deux listes :
  - la première contient les éléments inférieurs au pivot,
  - la seconde contient les éléments inférieurs au pivot.
- On trie alors récursivement ces deux sous-liste.
- On renvoie ces deux sous-listes triées avec la pivot au milieu.

⋆ Réalisation

Voici une réalisation très simple

```
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 Le code proposé ici est très naïf : on utilise deux listes auxiliaires pour stocker les deux sous-listes. On peut aussi déplacer les éléments dans L pour obtenir le même résultat.

⋆ Preuve de la terminaison

Proposition IV.1 La fonction `tri` se termine.

Démonstration. On utilise encore une récurrence (forte) :

$$\mathscr{P}(n) : \text{" si } len(L) = n \text{ alors la fonction tri se termine"}$$

c'est évident pour $n = 0$ et $n = 1$.

Si n est un entier vérifiant que $P(k)$ est vraie pour tout les $k < n$. Alors, on trouve le pivot et les deux sous listes `Lpetit` et `Lgrand` ont une longueur strictement inférieure à n, donc on peut appliquer l'hypothèse de récurrence pour affirmer que la fonction se termine sur ces deux sous-listes. La fonction tri se termine alors. ■

⋆ Preuve de la correction

Proposition IV.2 La fonction `tri` est correcte.

Démonstration. On utilise encore une récurrence (forte) :

$$\mathscr{P}(n) : \text{" si } len(L) = n \text{ alors la fonction tri renvoie le bon résultat."}$$

C'est évident pour $n = 0$ et $n = 1$.

Si n est un entier vérifiant que $P(k)$ est vraie pour tout les $k < n$. Alors, on trouve le pivot et les deux sous listes `Lpetit` et `Lgrand` ont une longueur strictement inférieure à n, donc on peut appliquer l'hypothèse de récurrence pour affirmer que la fonction `tri` renvoie bien deux sous-listes triées. Puisque le pivot est bien placé au bon endroit, la liste obtenue est bien triée. ■

⋆ Complexité

> **Proposition IV.3** Dans le pire des cas, la complexité de l'algorithme de tri rapide est $C(n) = O(n^2)$.
> Dans le meilleur des cas, la complexité de l'algorithme de tri rapide est $C(n) = O(n\log_2(n))$.

Démonstration. Si on note n la taille de la liste, et k le nombre d'éléments de la liste inférieur au pivot, alors on a la relation :

$$C(n) = n + C(k) + C(n-k)w$$

puisqu'on a besoin de n comparaisons pour construire les deux sous-liste et que l'on doit se rappeler sur chaque sous-liste.

Le pire des cas est obtenu si la liste est déjà triée ou si elle est triée dans le mauvais sens : dans ce cas : $k = 1$ ou $k = n-1$. On a alors :

$$C(n) = n + C(n-1)$$

et donc $C(n) = \underset{n\infty}{O}(n^2)$.

Le meilleur des cas est lorsque le pivot est au milieu, ie $k = \frac{n}{2}$:

$$C(n) = n + 2C\left(\frac{n}{2}\right)$$

en écrivant $n = 2^k$, cela donne :

$$C\left(2^k\right) = 2^k + 2C\left(2^{k-1}\right)$$

On obtient alors comme précédemment :

$$C\left(2^k\right) = k2^k$$

et donc $C(n) = O(n\log_2(n))$. ■

Malgré son nom, le tri rapide est donc *théoriquement* un tri plutôt lent : dans le pire des cas, sa complexité est quadratique, donc plutôt mauvaise.
On voit ici que ce qui est intéressant est en fait la complexité en moyenne, qui se calcule en mettant un modèle probabiliste sur les valeurs de la liste. Généralement, on considère que la liste contient une permutation aléatoire des entiers $[\![1,n]\!]$. Par un calcul de probabilité, on en déduit alors, l'espérance du nombre de comparaisons nécessaires pour trier la liste. Cette notion est hors-programme. Les calculs mathématiques nécessaires pour obtenir la complexité ne sont pas simples.
On admettra qu'en moyenne, le tri rapide a une complexité en $O(n\log_2(n))$.
En pratique, on choisit le pivot aléatoirement pour éviter de se retrouver dans le pire des cas.

⋆ Test sur un exemple

Voici une illustration sur un exemple :

```
L: [1, 14, 5, 4, 12, 13, 11, 8, 2, 3, 7, 6, 9, 10]
tri de : [1, 14, 5, 4, 12, 13, 11, 8, 2, 3, 7, 6, 9, 10]
le pivot est: 10
les deux listes à trier sont: [1, 5, 4, 8, 2, 3, 7, 6, 9]  et
    [14, 12, 13, 11]
tri de : [1, 5, 4, 8, 2, 3, 7, 6, 9]
le pivot est: 9
les deux listes à trier sont: [1, 5, 4, 8, 2, 3, 7, 6]  et  []
tri de : [1, 5, 4, 8, 2, 3, 7, 6]
le pivot est: 6
les deux listes à trier sont: [1, 5, 4, 2, 3]  et  [8, 7]
tri de : [1, 5, 4, 2, 3]
le pivot est: 3
les deux listes à trier sont: [1, 2]  et  [5, 4]
tri de : [1, 2]
le pivot est: 2
les deux listes à trier sont: [1]  et  []
tri de : [1]
tri de : []
tri de : [5, 4]
le pivot est: 4
les deux listes à trier sont: []  et  [5]
tri de : []
tri de : [5]
tri de : [8, 7]
le pivot est: 7
les deux listes à trier sont: []  et  [8]
tri de : []
tri de : [8]
tri de : []
tri de : [14, 12, 13, 11]
le pivot est: 11
les deux listes à trier sont: []  et  [14, 12, 13]
tri de : []
tri de : [14, 12, 13]
le pivot est: 13
les deux listes à trier sont: [12]  et  [14]
tri de : [12]
tri de : [14]
L: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

V Comparatif des algorithmes de tris

Voici un bilan des différents algorithmes de tris que l'on a étudié :

nom	complexité pire des cas	complexité meilleur cas	complexité moyenne
Sélection	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion	$O(n^2)$	$O(n)$	$O(n^2)$
Fusion	$O(n\log_2(n))$	$O(n\log_2(n))$	$O(n\log_2(n))$
Quicksort	$O(n^2)$	$O(n\log_2(n))$	$O(n\log_2(n))$

TABLE 4.1 – Comparatif des algorithmes de tris

Voici ce que l'on peut déduire de ce tableau :
- Les tris par sélection est plutôt lent. Attention : c'est celui qui fait le moins d'échange dans la liste, il a donc un intérêt lorsque l'opération qui prends du temps est l'échange dans la liste.
- Le tri par insertion est le meilleur dans le cas le plus favorable. Il est donc à réserver au cas où la liste est presque déjà triée.
- Le tri fusion est le plus rapide dans tous les cas. Son principal défaut est qu'il demande beaucoup de place en mémoire.
- Le tri rapide est très rapide dans le cas moyen mais lent dans des cas particulier. C'est souvent le tri par défaut, mais il faut l'éviter si le pire des cas peut se produire.

VI Recherche de la médiane

La **médiane** d'une liste de nombre `L = [L[0], ... ,L[n-1]]` contenant un nombre impair d'éléments est défini comme l'élément x de la liste tel que :

$$\begin{aligned} &\mathrm{Card}\{i \in [\![0, n-1]\!] \,|\, \mathtt{L[i]} < x\} = \left\lfloor \frac{n}{2} \right\rfloor \\ \text{et donc } &\mathrm{Card}\{i \in [\![0, n-1]\!] \,|\, \mathtt{L[i]} > x\} = \left\lfloor \frac{n}{2} \right\rfloor \end{aligned}$$

Autrement dit, il y a autant d'éléments inférieurs à x que supérieur à x dans la liste.

La médiane peut s'obtenir en triant la liste, puis en choisissant l'élément d'indice : $\lfloor \frac{n}{2} \rfloor$ (en partant de 0, donc le « $\lfloor \frac{n}{2} \rfloor + 1$-ième élément »).

Lorsque la liste est constituée d'un nombre pair d'élément, la médiane est obtenue en triant la liste par ordre croissant, puis en considérant la moyenne des deux éléments centraux :

$$\frac{1}{2}\left(\mathtt{L}\left\lfloor \frac{\mathtt{n}}{2} - 1 \right\rfloor + \mathtt{L}\left\lfloor \frac{\mathtt{n}}{2} \right\rfloor\right)$$

R La médiane est une mesure statistique permettant de donner une information de position sur une série statistique. C'est-à-dire qu'elle indique une mesure représentative de la série (comme la moyenne).

Elle est plus robuste aux erreurs que la moyenne, car si on ajoute une très forte valeur à la série, on modifie beaucoup la moyenne mais très peu la médiane.

Plus généralement, étant donné une liste L, on souhaite chercher le k-ième élément dans l'ordre croissant. Autrement dit, on cherche l'élément $j \in L$ tel que :

$$k - 1 = \text{Card}\left(\{x \in \text{L} | x < j\}\right)$$

Bien entendu, on peut trier la liste entièrement puis prendre le k-ième élément. On cherche à faire plus rapidement.

Remarquons déjà qu'on peut supposer $k \leqslant \left\lfloor \frac{n}{2} \right\rfloor$, si ce n'est pas le cas, il suffit de classer la liste « *à l'envers* », puis de prendre l'élément $n-k$. Le cas critique est ainsi la recherche de la **médiane**.

On reprend les algorithmes de tri pour voir quelle méthode on peut adapter :

tri par sélection Il suffit de s'arrêter une fois que l'on a classé k termes.

tri par insertion On ne peut pas l'adapter facilement à la recherche du k-ième élément.

tri fusion On ne peut pas l'adapter facilement à la recherche du k-ième élément.

tri rapide s'adapte facilement : si le pivot est à la place k, on a terminé, sinon, on ne trie que la partie intéressante.

Voici une adaptation du tri par sélection à la recherche de l'élément pos dans l'ordre croissant :

```
def mediane_selection(L, pos):
  n = len(L)
  for i in range(pos+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[pos]
```

Voici une adaptation du tri rapide à la recherche de l'élément pos dans l'ordre croissant :

```
def mediane(L, pos):
    """
    entrée: L = list
            pos = position de l'élement
            recherché dans la liste triée
    sortie: l'élément recherché
    """
    if len(L) <= 1 :
        return L[0]
    print("liste:", L,"on cherche l'élément", pos)
    pivot = L.pop()
    print("pivot:", pivot)
    Lpetit = [ x for x in L if x <= pivot]
    k = len(Lpetit)  # position du pivot
    if k == pos :
        print(" on a fini")
        return pivot
```

```
    elif pos < k:
        print(" on se relance sur les plus petits")
        return mediane(Lpetit, pos)
    else:
        Lgrand = [ x for x in L if x > pivot]
        print(" on se relance sur les plus grands")
        return mediane(Lgrand, pos-k-1)
```