

Cours d'informatique (MP)

Arnaud GIRAND

18 juillet 2021

Ce document est placé sous licence CC BY-NC-SA 3.0.

Table des matières

I	Méthodes numériques	5
1.-	Recherche de zéros de fonctions continues	5
2.-	Méthode d'Euler	7
II	Structures de données	11
1.-	La classe <code>list</code>	11
2.-	La classe <code>numpy.array</code>	13
3.-	Piles	14
III	Récurtivité	17
1.-	C'est quoi?	17
2.-	Zoologie récursive	18
IV	Algorithmes de tri	23
1.-	Tri par insertion	23
2.-	Tri rapide	25
3.-	Tri fusion	26
V	Kit de survie en terre SQL	29
1.-	De quoi cause-t-on?	29
2.-	Interrogation d'une base de données	30
	Aide-mémoire de complexité	33

Chapitre I

Méthodes numériques

1. Recherche de zéros de fonctions continues

Dans ce paragraphe, nous nous fixons une fonction f continue sur un segment $[a, b]$ et à valeurs réelles. Nous faisons l'hypothèse que la fonction change de signe en un certain point $c \in]a, b[$ dont nous souhaitons obtenir une valeur approchée ; cette hypothèse peut être traduite par la condition suivante :

$$f(a)f(b) < 0. \quad (\mathbf{E:1.1})$$

a) Recherche par dichotomie

On définit par récurrence trois suites $(a_n)_n$, $(b_n)_n$ et $(c_n)_n$ selon le procédé suivant : posons $a_0 = a$, $b_0 = b$ et $c_0 = \frac{a+b}{2}$, puis, pour tout $n \geq 0$

- ▷ si $f(a_n)f(c_n) < 0$, on pose $a_{n+1} = a_n$ et $b_{n+1} = c_n$;
- ▷ dans le cas contraire, on pose $a_{n+1} = c_n$ et $b_{n+1} = b_n$.

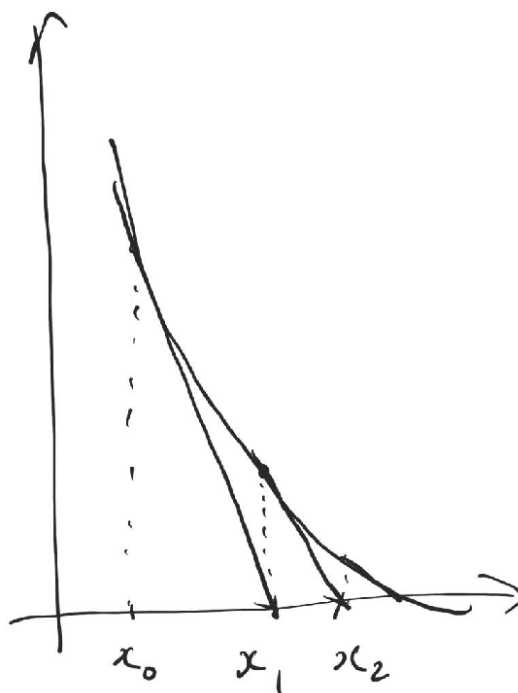
Enfin, dans tous les cas, on pose :

$$c_{n+1} = \frac{a_{n+1} + b_{n+1}}{2}.$$

On resserre donc petit à petit une "fenêtre" autour du point d'annulation de f , ce qui entraîne que la suite $(c_n)_n$ converge vers un point d'annulation de f . On peut implémenter cette procédure en python via le code suivant.

```
def dichotomie(f, a, b, p):
    if f(a)*f(b) > 0:
        raise ValueError("f ne s'annule pas")
    while b-a > 10**(-p):
        c = (a+b)/2
        if f(c) == 0:
            return c
        if f(a)*f(c) < 0:
            b = c
        else:
            a = c
    return c
```

b) Méthode de Newton



On suppose désormais que f est dérivable et sympathique ; elle sera alors localement croissante ou décroissante au voisinage de c (puisqu'on vous dit qu'elle est sympathique). Si on fixe x_0 pas trop loin de c , la tangente de f en x_0 va "pointer" vers le point d'annulation c au sens suivant : le point d'intersection x_1 entre cette dernière et l'axe des abscisses sera plus proche de c que x_0 . Ce point x_1 doit vérifier :

$$0 = f(x_0) + (x_1 - x_0)f'(x_0)$$

et donc, si $f'(x_0)$ a la gentillesse de ne pas s'annuler trop fort $x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$.

À supposer que nous habitons dans le monde magique des fonctions gentiment localement monotones sans annulation intempestive de leurs fonctions dérivées, nous pouvons donc itérer ce procédé via la formule de récurrence :

$$\forall n \in \mathbb{N}, x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

et cette suite convergera vaguement vers le point c tel que $f(c) = 0$. En python, cela donne une procédure du style décrit *infra*.

```
def newton(f, fp, x0, N):
    x = x0
    for i in range(N):
        x = x - f(x)/fp(x)
    return x
```

Notons que cette fonction prend en argument la fonction f et sa dérivée fp et que nous avons utilisé un nombre d'itérations maximal comme condition d'arrêt ; cette méthode est en effet assez peu fiable dans le cas général et nous souhaitons éviter une boucle infinie ; il convient de toujours vérifier les résultats qu'elle renvoie.

2. Méthode d'Euler

a) Équations différentielles ordinaires

Ce paragraphe a pour but de fournir au lecteur un kit de survie en territoire différentiel non linéaire. Si cela vous effraie, dites vous que vous le faites en permanence en physique...

Définition I.1. Soit I un intervalle de \mathbb{R} . On appelle **équation différentielle** (ordinaire) toute équation de la forme

$$\forall t \in I, \quad y' = f(t, y(t))$$

où :

- ▷ $f : I \times \mathbb{R} \rightarrow \mathbb{R}$ est une fonction continue de deux variables ;
- ▷ $y \in \mathcal{D}^1(I)$ est l'inconnue de l'équation.

Notation. Pour abrégé, on écrira souvent " $y' = f(t, y)$ ".

Exemple I.1.

- ▷ Si f est de la forme $(t, y) \mapsto a(t)y + b(t)$ avec $a, b \in \mathcal{C}^0(I)$, on retombe sur les équation différentielle linéaire d'ordre 1 vues en MPSI ;
- ▷ pour $f(t, y) \mapsto \sqrt{y}$, on obtient l'équation **non linéaire** $y' = \sqrt{y}$.

Ce qui suit est une version légèrement plus forte, mais toujours relativement faible, du théorème de Cauchy–Lipschitz vu pour les équations linéaires en MPSI.

Théorème I.1 (Cauchy–Lipschitz faible).

Si $f : I \times \mathbb{R} \rightarrow \mathbb{R}$ est une fonction continue de deux variables lipschitzienne par rapport à sa deuxième variable, alors le système :

$$\begin{cases} y' = f(t, y) \\ y(t_0) = y_0 \end{cases}$$

admet une unique solution pour tout couple de conditions initiales $(t_0, y_0) \in I \times \mathbb{R}$.

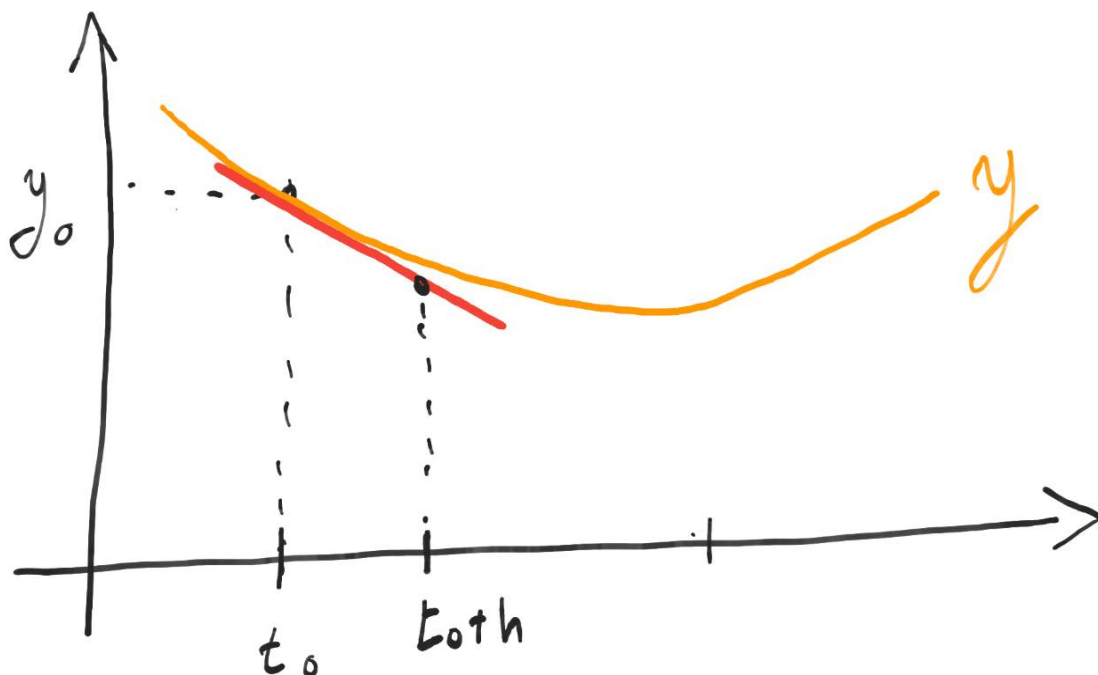
b) Résolution approchée par méthode d'Euler

L'objectif de ce paragraphe est de donner une méthode permettant, étant donné une paire de conditions initiales (t_0, y_0) et une équation différentielle $y' = f(t, y)$ vérifiant les hypothèses du théorème de Cauchy–Lipschitz, d'approcher la solution du problème de Cauchy correspondant. Pour fixer les idées, posons $I = [t_0, t_0 + T]$, avec $T \in \mathbb{R}_+^*$ et cherchons à déterminer une approximation y de la solution sur I de :

$$\begin{cases} y' = f(t, y) \\ y(t_0) = y_0 \end{cases} .$$

Commençons par fixer un pas $h > 0$, par exemple $h = \frac{T}{N}$ avec N suffisamment grand, et notons que, au voisinage de t_0 , la formule de Taylor–Young s’écrit :

$$\begin{aligned} y(t) &= y(t_0) + (t - t_0)y'(t_0) + o_{t_0}(t - t_0) \\ &= y_0 + (t - t_0)f(t_0, y_0) + o_{t_0}(t - t_0). \end{aligned}$$



On peut donc raisonnablement approximer la fonction y au voisinage de t par l’expression :

$$y(t) \cong y_0 + (t - t_0)f(t_0, y_0)$$

et donc, si h est choisi assez petit, on a :

$$y(t_0 + h) \cong y_0 + hf(t_0, y_0).$$

Nous pouvons de fait approcher la fonction y sur le segment $[t_0, t_0 + h]$ par la fonction affine $t \mapsto y_0 + (t - t_0)f(t_0, y_0)$. Ceci revient géométriquement à substituer localement à la courbe de y sa tangente en t_0 .

En itérant ce procédé, nous pouvons approcher y sur I par la fonction affine par morceaux reliant les points (t_n, y_n) définis comme suit (à partir des termes initiaux t_0 et y_0 :

$$\forall n \in \mathbb{N} \text{ tel que } 0 \leq n \leq \left\lfloor \frac{T}{h} \right\rfloor - 1, \quad \begin{cases} t_{n+1} = t_0 + (n+1)h \\ y_{n+1} = y_n + hf(t_n, y_n) \end{cases}.$$

Cette fonction affine par morceaux est définie par la formule

$$t \mapsto y_n + (t - t_n)f(t_n, y_n)$$

sur le segment $[t_n, t_{n+1}]$.

En python, tout ceci peut se réaliser de la façon suivante.


```
from numpy import *  
  
def euler(f,t0,y0,T,N):  
    h=T/N  
    X=linspace(t0,t0+T,N+1)  
    Y=[y0]  
    for k in range(N):  
        Y=Y+[Y[k]+h*f(X[k],Y[k])]  
    return (X,Y)
```

La complexité de cette fonction est aisée à établir : le paramètre variable d'intérêt ici est N , qui contrôle la précision. Comme l'exécution de f ne dépendra pas de celui-ci en première approximation, on peut conclure que la fonction `euler` a une complexité en $\mathcal{O}(N)$.

Chapitre II

Structures de données

Le but de ce chapitre est de présenter plusieurs façons d'**organiser** des données informatiques tout en restant en capacité d'y **accéder** de façon raisonnable.

1. La classe `list`

Commençons par noter que la classe `list` de `python` ne répond pas à la définition classique d'une liste en informatique théorique. Il s'agit plutôt d'une classe hybride liste/tableau dans la mesure où la taille d'une instance n'est pas prescrite (liste) mais où il reste possible d'accéder à chaque élément de la dite instance en temps constant (tableau).

Lorsque l'on demande à `python` de créer une instance de la classe `list`, *via* une commande du type `L=[]` ou `L = [1,2, "lapin"]`, celui-ci alloue une zone mémoire suffisamment grande à cette dernière pour permettre de l'étendre si besoin : cette classe est donc idéale pour manipuler des collections d'objets évolutives dont la taille n'est pas initialement prévue.

✘ **ATTENTION** : Si `L` est une liste, un appel à `S=L` ne créera **pas** une liste `S` copie conforme de `L` mais simplement un pointeur vers cette dernière. Toute modification opérée sur l'une ou l'autre des listes impactera donc sa consœur.

a) Manipulation des listes

◇ Attributs

- ▷ Étant donné une liste `L`, on a accès à l'attribut `len(L)` qui renvoie (en temps constant) le nombre d'éléments contenus dans `L`.

▮► **Exemple II.1.** `len([1,2,3])` renvoie 3.

- ▷ On a également accès au maximum et au minimum d'une liste `L` via `max(L)` et `min(L)`. Ces fonctions ont une complexité linéaire en `len(L)`.
- ▷ On peut tester (en temps linéaire en `len(L)`) l'appartenance d'un objet `e` à une liste `L` via `e in L`.

▮► **Exemple II.2.** Un appel à `2 in [1,0,4]` renverra `False`.

◇ Éléments, sous-listes

- ▷ Si i est un indice compris entre 0 et $\text{len}(L)-1$, l'élément situé en position i de la liste L peut être obtenu *via* un appel à $L[i]$.

▮► **Exemple II.3.** Si $L=[0,-1,"carotte"]$, un appel à $L[1]$ renverra -1 .

- ▷ Si i, j sont deux indices compris entre 0 et $\text{len}(L)-1$, un appel à $L[i:j]$ renverra, en $\mathcal{O}(j-i)$, une copie de la liste $[L[i], L[i+1], \dots, L[j-1]]$.

▮► **Exemple II.4.** Si $L=[1,2,3,4,5]$, un appel à $L[1:3]$ renverra $[2,3]$.

Notons que ceci crée une nouvelle liste ne partageant **pas** une adresse mémoire avec la liste initiale. On pourra donc modifier cette dernière sans crainte d'impacter une extraction préalable.

✂ **Remarque II.1.** Un appel à $L[:j]$ est équivalent à $L[0:j]$; de même, $L[i:]$ renverra $L[i:\text{len}(L)]$.

✂ **Remarque II.2.** Ces deux commandes peuvent être utilisées pour modifier une liste L . Nous encourageons le lecteur à exécuter le bloc de code suivant.

```
L = [0, 1, 2, 3, 4, 5, 6]
L[1:4] = [1]
```

◇ Modification

- ▷ Si L est une liste et que e est un objet, les commandes $s.append(e)$ et $s+=[e]$ ajouteront l'objet e en queue de liste, en temps constant.
- ▷ Si S et L sont deux listes, un appel à $S+L$ renverra une liste composée des éléments de S suivis de ceux de L en temps linéaire en $\text{len}(L)$.
- ▷ Si L est une liste et n un entier naturel, un appel à $n*L$ renverra une liste formée de n copies successives de L .

▮► **Exemple II.5.** Un appel à $2*[1,3]$ renverra $[1,3,1,3]$.

- ▷ Si i, j sont deux indices compris entre 0 et $\text{len}(L)-1$, un appel à `del L[i:j]` supprimera les éléments $L[i], \dots, L[j-1]$ de la liste L . La complexité temporelle d'une telle opération est linéaire en $j-i$.
- ▷ Si L est une liste, e un objet et i un indice compris entre 0 et $\text{len}(L)-1$, un appel à $L.insert(i,e)$ modifiera L en y ajoutant e en position i , décalant d'éventuels éléments postérieurement placés. Une telle opération coûte $\mathcal{O}(\text{len}(L))$.
- ▷ Si L est une liste et i un indice compris entre 0 et $\text{len}(L)-1$, un appel à $L.pop(i)$ supprimera (en complexité linéaire en $\text{len}(L)$) l'élément $L[i]$ et le renverra. Un appel à $L.pop()$ fera de même **en temps constant** avec le dernier élément de L .

▮► **Exemple II.6.** Si $L=[1,2,"a"]$, un appel à $L.pop()$ renverra "a" et transformera L en $[1,2]$.

b) Conclusion

◇ Avantages

La classe `list` est **flexible**, permettant de **modifier la taille** des instances, de combiner plusieurs instances ou de pratiquer des extractions en un temps raisonnable. Ceci la rend idéale pour la manipulation de plages de données de taille variable (*e.g* modélisation de suites, méthodes numériques itératives, ...).

◇ Inconvénients

La complexité spatiale de la classe `list` n'est pas, par nature, finement contrôlée. Ceci peut entraîner la création d'objets de taille trop importante pour notre système (fuite mémoire) si l'on n'est pas vigilant. Par ailleurs, n'oublions pas que plusieurs fonctions créent de nouvelles listes (par exemple les extractions) : en abuser ne peut qu'accentuer le problème.

2. La classe `numpy.array`

Dans tout ce paragraphe, nous considérerons l'instruction suivante exécutée.

```
from numpy import *
```

La classe `array` du module `numpy` nous fournit une classe de type tableau en `python`. La principale différence entre cette classe et la classe `list` est le fait que les instances créées sont de taille fixe, prescrite par le constructeur. Ceci permet une gestion fine de l'usage fait de la mémoire vive par nos algorithmes, et donc d'en borner la complexité spatiale.

Pour créer une instance du type `numpy.array`, on dispose de plusieurs options.

- ▷ Si `L` est une liste, un appel à `array(L)` renverra un tableau construit à partir de celle-ci.
 - ✘ **ATTENTION** : Contrairement à la classe `list`, la classe `numpy.array` exige que le type des éléments d'une instance soit constant.
- ▷ Un appel à `empty((n,m))` renverra un tableau "vide" (rempli de différentes déjections mémoire) de taille `(n,m)` et dont les entrées seront de type `float`. Il est possible de prescrire le type des entrées via un appel à `empty((n,m), type)`. On peut également créer un tableau à 1 ligne et `n` colonnes via `empty(n)`. Le lecteur tordu sera sans doute extatique d'apprendre qu'il est également possible de construire des tableaux ayant plus de deux dimensions via une syntaxe du type `empty((n1,n2,...,nk))`.
- ▷ Si le lecteur préfère voir son tableau rempli de 1 ou de 0, il peut utiliser les constructeurs `ones` et `zeros`, dont la syntaxe est identique à celle de `empty`.

a) Manipulation des tableaux

◇ Attributs

Si `T` est un tableau, on a accès aux attributs suivant en temps constant :

- ▷ `T.ndim` renvoie le nombre de dimensions de `T` ;
- ▷ `T.shape` renvoie les dimensions de `T` sous la forme d'un k -uplet ;

- ▷ `T.size` renvoie le nombre d'éléments de `T` ;
- ▷ `len(T)` renvoie le nombre de lignes de `T` ;
- ▷ on accède aux éléments d'un tableau suivant la même syntaxe que pour la classe `list`.

▣► **Exemple II.7.** Si `T=array([[1,2],[3,4]])`, un appel à `T[0][1]` renverra `2`.

Il est possible de les modifier suivant la même syntaxe que pour la classe `list`.

De plus, il est possible d'accéder en temps linéaire (en `T.size`) aux valeurs extrêmes du tableau via `T.max()` et `T.min()`. On peut également tester (avec la même complexité temporelle) l'appartenance d'un objet `x` au tableau `T` via l'instruction `x in T`.

◇ Opérations

- ▷ Il est possible d'effectuer des combinaisons linéaires (au sens matriciel du terme) de tableaux de même dimension.

▣► **Exemple II.8.** Si `T` est le tableau correspondant à la liste `[1,3,4]`, un appel à `T + 2*T` renverra le tableau `array([3,9,12])`.

- ▷ Si `T` est un tableau et que `x` est un objet de même type que les éléments de `T`, un appel à `T+x` renverra le tableau obtenu en ajoutant `x` à chacune des entrées de `T`.

b) Conclusion

◇ Avantages

La classe `numpy.array` permet de contrôler finement la taille des objets manipulés, et donc la complexité spatiale des algorithmes. Elle dispose de plus d'une structure linéaire native, qui la rend particulièrement adaptée à la modélisation d'objets matriciels.

◇ Inconvénients

La taille d'un tableau étant fixée, il est impossible de modifier sa structure : concaténation, scission et autres joyeusetés sont proscrits. Par ailleurs, le type des entrées doit être constant, ce qui rigidifie encore la manipulation de cette classe (mais la rend plus prévisible).

3. – Piles

a) C'est quoi ?

On appelle **pile** toute structure de donnée de type LIFO (Last In First Out) ; *i.e* toute structure dans laquelle le dernier élément ajouté est le seul accessible en temps constant. Cela signifie que l'utilisateur n'a à sa disposition que deux opérations :

- ▷ l'**empilage** (`push`), qui consiste à ajouter un élément en queue de pile ;

▷ le **dépilage**, (**pop**) consistant à retirer de la pile son élément terminal.

Cette structure de donnée présente plusieurs avantages : elle est de taille flexible, doté d'un ordre de traitement clair et prévisible, et permet des ajouts et suppressions en temps constant. La contrainte majeure, et elle est de taille, est paradoxalement que l'ordre de traitement est clair, donc immuable.

b) Une implémentation possible

Un exemple de classe "pile" en python est donné dans ce paragraphe. Nous rappelons que les subtilités de la déclaration d'une classe ne sont pas exigibles de notre lecteur, et qu'il lui sera tout à fait possible de "simuler" une pile en se restreignant aux méthodes `append` et `pop` de la classe `list`.

```
class pile:
    """ Classe pile pour python """

    def __init__(self):
        """ Constructeur de pile vide """
        self.lst=[]

    def empty(self):
        """ Vérifie si une pile est vide """
        return self.lst==[]

    def push(self,x):
        """ Empiler x sur la pile """
        self.lst.append(x)

    def pop(self):
        """ Dépilage """
        if self.empty():
            raise ValueError("vide")
        return self.lst.pop()
```

c) Application : notation polonaise inverse

On appelle **notation polonaise inverse** la notation post-fixée des opérations algébriques. Elle dérive des travaux du mathématicien, philosophe et logicien (polonais) Jan Łukasiewicz (1878—1956).


▮► **Exemple II.9.** L'opération $12 \times (6 + 11)$ sera notée $12611 + \times$. On lit une telle expression de droite à gauche en appliquant chaque opération aux deux entiers (ou résultats d'opérations intermédiaires) à sa gauche. Ici, on doit appliquer \times à $611 +$ et 12 , *i.e* à 17 et 12 .

Cette approche possède l'avantage de ne pas nécessiter de parenthèse : l'ordre des opérations est non ambigu et s'apparente au dépilage d'une pile.

Pour implémenter ceci en python, nous définissons une fonction `comp` permettant de calculer le résultat d'une suite d'opérations post-fixées (présentée sous la forme d'une pile `p`) **supposée correcte**.

```
def comp(p):
    val=pile()
    while not p.empty():
        x=p.pop()
        if x=='+' :
            a=val.pop()
            b=val.pop()
            val.push(a+b)
        elif x=='*' :
            a=val.pop()
            b=val.pop()
            val.push(a*b)
        elif x=='/' :
            a=val.pop()
            b=val.pop()
            val.push(a/b)
        elif x=='-' :
            a=val.pop()
            b=val.pop()
            val.push(a-b)
        else :
            val.push(x)
    return val.pop()
```

Nous encourageons notre lecteur à visualiser l'action de cet algorithme à l'aide d'un arbre sur un exemple.

 **Exercice II.1.** Écrire une fonction python vérifiant la correction d'une pile représentant un calcul post-fixé.

Chapitre III

Récurtivité

"Pour comprendre la récurtivité, il faut d'abord comprendre la récurtivité."

1. C'est quoi ?

Notre lecteur préféré (et les autres aussi) est sans nul doute à ce stade de son éducation familier avec le concept de récurrence en mathématiques, qui permet de définir des objets indexés par \mathbb{N} en fixant une valeur initiale et en donnant un procédé permettant, étant donné un élément de la famille, de calculer son successeur (hérédité).

Prenons par exemple la suite $u \in \mathbb{R}^{\mathbb{N}}$ définie par $u_0 = 3$ et la relation de récurrence suivante :

$$\forall n \in \mathbb{N}, \quad u_{n+1} = \sqrt{1 + u_n}.$$

Pour calculer, étant donné un entier n , le terme u_n à l'aide de `python`, nous pouvons faire usage de la fonction suivante (pour peu que le module `numpy` soit importé).

```
def u(n):
    res=3
    for i in range(n):
        res=sqrt(1+res)
    return res
```

Cet algorithme fait usage d'une boucle pour obtenir le résultat voulu : il itère donc un même procédé n fois afin de parvenir à ses fins. On appelle, fort originalement, un tel procédé une **fonction itérative** (ou un algorithme itératif selon humeur).

On pourrait cependant envisager une autre façon de procédé, plus proche dans sa philosophie de la définition par récurrence donnée *supra* pour notre suite.

```
def uRec(n):
    if n==0:
        return 3
    return sqrt(1+uRec(n-1))
```

Cette fonction est composée de deux "blocs", qu'il est essentiel de bien identifier :

- ▷ un **cas de base** (correspondant à l'initialisation de la récurrence sous-jacente) ;
- ▷ un **appel récursif**, où la fonction s'appelle elle-même.

Une telle fonction est, sans surprise la non plus, qualifiée de **récursive**. Un appel à celle-ci sera traité par `python` à l'aide d'une pile, de façon similaire à ce que nous avons vu pour l'interprétation des opérations en notation polonaise inverse.

 **Exercice III.1.** Décrire les étapes d'un appel à `uRec(3)`.

L'usage de fonctions récursives présente un avantage non négligeable : elles sont en générales plus "compactes" que leurs consœurs itératives, et donc plus aisées à déboguer. Nous verrons également par la suite que certains problèmes se prêtent naturellement à une résolution récursive.

✘ ATTENTION : cela ne signifie pas que l'approche récursive soit meilleure en tous cas que l'usage de fonctions itératives. En particulier :

- ▷ le cas de base est absolument nécessaire au bon fonctionnement d'une fonction récursive, et toute erreur, même minime, dans ce dernier, entraînera une chaîne infinie d'appels récursifs (ce qui est long) ;
- ▷ `python` limite le nombre d'appel récursifs à 1000, afin de limiter le risque de suite infinie d'appels. Il est toutefois possible de modifier ceci ; le bloc d'instructions suivant la fixe par exemple à 2000.

```
import sys
sys.setrecursionlimit(2000)
```

Il est toutefois déconseillé de procéder à une telle modification lorsqu'elle n'est pas absolument nécessaire.

2. Zoologie récursive

Dans ce paragraphe, nous présentons une série d'algorithmes récursifs et les comparons à leurs équivalents itératifs lorsque cela est pertinent. Loin de se vouloir exhaustive, cette liste a pour objectif d'offrir un lecteur un panorama des différents atouts et inconvénients de l'approche récursive.

a) Calcul de factorielles

L'algorithme itératif suivant permet, étant donné un entier n , de calculer $n!$.

```
def fact(n):
    f=1
    for i in range(n):
        f*=(i+1)
    return f
```

Sa complexité est en $\mathcal{O}(n)$.

✘ ATTENTION : ceci signifie que cet algorithme est de complexité **exponentielle** en la taille de l'entier n en mémoire, qui correspond peu ou prou au nombre de chiffres de son écriture binaire, soit vaguement $\lg(n)$.

Nous proposons ensuite la fonction récursive suivante, en espérant qu'elle fonctionne, pour réaliser le même calcul.

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

Loin de plaisanter *supra*, nous effleurons l'un des problèmes inhérents à l'approche récursive : il n'est absolument pas acquis *a priori* que notre algorithme s'arrêtera un jour (**terminaison**) et rende le bon résultat (**correction**). Il nous faut donc le démontrer.

Proposition III.1. Étant donné un entier naturel n , un appel à `factR(n)` termine et renvoie la valeur $n!$.

Démonstration. Sans surprise, nous allons démontrer ceci par récurrence sur l'entier naturel n .

- Si $n = 0$, l'instruction conditionnel `if` assure que nous renvoyons $1 = 0!$.
- Supposons la propriété vraie à un certain rang n . Un appel à `factR(n+1)` ne déclenchera pas l'instruction conditionnelle (car $n \geq 0$) et donc renverra $(n+1)*\text{factR}(n)$, qui termine par hypothèse de récurrence et renvoie

$$(n + 1) \times n! = (n + 1)!,$$

d'où l'hérédité.

Notre assertion étant initialisée et héréditaire, nous pouvons pousser un soupir collectif de soulagement : elle est vraie. \square

Concernant la complexité, si on note U_n le nombre d'opérations nécessaires au calcul de $n!$ via `factR`, on a la relation de récurrence :

$$\forall n \geq 0, \quad U_{n+1} = 1 + U_n$$

car on fait une multiplication par appel récursif. On obtient donc une complexité en $\mathcal{O}(n)$, similaire à celle de l'algorithme itératif `fact`.

b) Suite de Fibonacci

On définit une suite récurrente d'ordre 2 $(F_n)_n$ via $F_0 = 0$, $F_1 = 1$ et :

$$\forall n \geq 0, \quad F_{n+2} = F_n + F_{n+1}.$$

Un algorithme itératif calculant les termes successifs de cette suite peut être donné par la fonction suivante.

```
def fibo(n):
    F=0
    G=1
    for i in range(n):
        F, G=G, F+G
    return F
```

La complexité d'un appel à `fibonacci(n)` est en $\mathcal{O}(n)$, soit exponentielle en la taille de son argument.

Pour calculer récursivement les termes successifs de la suite $(F_n)_n$, on peut être tenté d'écrire la fonction suivante, qui "traduit" directement la relation de récurrence sous-jacente à cette dernière.

```

def fiboR(n):
    if n==0:
        return 0
    if n==1:
        return 1
    return fiboR(n-1)+fiboR(n-2)

```

Celle-ci accuse cependant un défaut **majeur** : elle comporte un double appel récursif. Ceci signifie que si C_n est le nombre d'opérations requis pour l'exécution de `fiboR(n)`, on a la relation :

$$\forall n \geq 0, \quad C_{n+1} = C_n + C_{n-1},$$

ce qui entraîne que $C_n = \mathcal{O}(\phi^n)$, avec $\phi = \frac{1+\sqrt{5}}{2}$. Notre fonction a donc une complexité bien supérieure à son homologue itérative, au point de la rendre totalement inutilisable en pratique.

Une façon de résoudre ce problème et d'obtenir une fonction de complexité $\mathcal{O}(n)$ est de faire usage d'une fonction auxiliaire gardant en mémoire deux termes consécutifs de la suite de Fibonacci.

```

def aux(n):
    if n==0:
        return [0,1]
    L=aux(n-1)
    return [L[1],L[0]+L[1]]

```

```

def fiboR2(n):
    return aux(n)[0]

```

✂ **Remarque III.1.** Le lecteur attentif remarquera que nous faisons usage d'une variable `L` dans la fonction `aux` afin d'éviter de résoudre notre problème d'appels récursifs multiples à l'aide... d'appels récursifs multiples.

✂ **Exercice III.2.** Démontrer la correction et la terminaison de la fonction `aux`.

c) Recherche dichotomique dans une liste triée

Étant donné une liste **supposée triée** `L` et un objet `x`, la fonction suivante renvoie un booléen qualifiant la présence de `x` dans `L` en effectuant une recherche dichotomique.

```

def dichot(x,L):
    if len(L)==0:
        return False
    k = len(L)//2
    if x==L[k]:
        return True
    if x < L[k]:
        return dichot(x,L[:k])
    return dichot(x,L[k+1:])

```

✎ **Exercice III.3.** Démontrer la correction et la terminaison de la fonction `dicho`.

Pour déterminer la complexité de cet algorithme, notons C_n le nombre (asymptotique) d'opérations requis pour trouver un objet x dans une liste L de taille $n \geq 0$. On a alors la relation de récurrence suivante :

$$\forall n \geq 0, \quad C_{n+1} = C_{\frac{n}{2}} + \frac{n}{2},$$

le second terme de la somme correspondant aux opérations nécessaires pour "recopier" la moitié de la liste via l'instruction `L[k+1:]` ou `L[:k]`. Pour simplifier les choses, supposons que la taille de notre liste est de la forme $n = 2^p$; on a alors :

$$\forall p \geq 1, \quad C_{2^p} = C_{2^{p-1}} + 2^{p-1}$$

et donc

$$\begin{aligned} \forall p \geq 1, \quad C_{2^p} &= \sum_{k=1}^{p-1} 2^k \\ &= 2^p - 1 \\ &= \mathcal{O}(2^p). \end{aligned}$$

Notre fonction a donc une complexité linéaire en la taille de la liste L .

✎ **Remarque III.2.** Il est possible d'écrire un algorithme de complexité logarithmique en évitant l'étape de recopie *via* l'usage d'une fonction auxiliaire.

```
def aux(x, L, i, j):
    if j-i==0:
        return False
    k = (i+j)//2
    print(i, j, k)
    if x==L[k]:
        return True
    if x < L[k]:
        return aux(x, L, i, k)
    return aux(x, L, k+1, j)

def dicho2(x, L):
    return aux(x, L, 0, len(L))
```

d) Exponentiation rapide

Pour calculer récursivement le flottant x^n avec x flottant et n entier naturel, on pourrait être tenté de faire usage de la fonction suivante.

```
def expo(x, n):
    if n==0:
        return 1
    return x*expo(x, n-1)
```

La complexité de cette dernière est hélas en $\mathcal{O}(N)$, *i.e* exponentielle en la taille de l'exposant. Ceci est problématique; une opération aussi courante ne saurait être mise hors de notre portée par de basses limitations techniques! Fort heureusement, nous disposons de l'algorithme d'**exponentiation rapide** (utilisé par python dans l'opérateur ******).

```
def expRap(x, n):
    if n==0:
        return 1
    if n%2==0:
        return expRap(x*x, n/2)
    return x*expRap(x*x, (n-1)/2)
```

Proposition III.2. La fonction `expRap` termine et est correcte.

Démonstration. Démontrons par récurrence forte sur $n \in \mathbb{N}$ que tout va bien.

- Si $n = 0$, la première instruction conditionnelle nous sauve.
- Si tout va bien à un certain rang $n \geq 0$, un appel à `expRap(x,n+1)` fera soit appel à `expRap(x*x,n/2)`, soit à `expRap(x*x,(n-1)/2)` qui terminent tous deux par hypothèse de récurrence (forte) et sont corrects pour la même raison. Un calcul rapide permet de vérifier la correction et de conclure la démonstration.

□

Pour estimer la complexité de cet algorithme, fixons nous dans le cas où n est de la forme 2^p et notons u_p le nombre (asymptotique) d'opérations requis pour exécuter `expRap(x, 2p)`. On a alors la relation de récurrence suivante :

$$\forall p \in \mathbb{N}, \quad u_{p+1} = u_p + 1$$

et donc $u_p = \mathcal{O}(p)$. La complexité de l'exponentiation rapide est de fait linéaire en la taille de l'exposant.

Chapitre IV

Algorithmes de tri

L'objet de ce chapitre est de **classer** des objets, rassemblés dans une **structure linéaire de donnée** (tableau ou liste) selon un **ordre** préétabli. Nous présentons différentes méthodes permettant d'y parvenir et évaluons leur **coût**.

1. Tri par insertion

a) Qu'est-ce ?

Le tri par insertion est un tri "naturel", souvent utilisé dans la vie courante. On commence par placer le premier élément au début de la liste, puis on compare chaque élément de la liste à ceux le précédant de façon à l'insérer "à sa place". Un tel procédé appliqué à la liste $L=[3,2,18,1,-1,0,12]$ passerait donc par les étapes illustrées *infra*.

```
>>> L=[3,2,18,1,-1,0,12]
>>> tri_insertion(L)
[2, 3, 18, 1, -1, 0, 12]
[2, 3, 18, 1, -1, 0, 12]
[1, 2, 3, 18, -1, 0, 12]
[-1, 1, 2, 3, 18, 0, 12]
[-1, 0, 1, 2, 3, 18, 12]
[-1, 0, 1, 2, 3, 12, 18]
```

On peut implémenter ce tri en langage python par la procédure suivante.

```
def triInsertion(L):
    n=len(L)
    for i in range(1,n):
        x=L[i]
        p=i
        while p>0 and L[p-1]>x:
            L[p]=L[p-1]
            p=p-1
        L[p]=x
```

✘ **ATTENTION** : Il s'agit bien de la d'une **procédure** : la liste L est **directement modifiée**, et notre algorithme ne renvoie aucun résultat.

b) Correction et terminaison

Lorsque l'on manipule un algorithme complexe, il est essentiel de se poser deux questions (outre celle de sa complexité, sur laquelle nous reviendrons incessamment) :

- ▷ l'algorithme s'arrête-t-il un jour (**terminaison**) ?
- ▷ L'algorithme donne-t-il le bon résultat (**correction**) ?

Dans le cas de la procédure `triInsertion`, la terminaison est immédiate : en effet, la boucle `for` possède un nombre fixé d'étapes et la boucle `while` est indexée par une suite strictement décroissante d'entiers naturels. Pour démontrer la correction, nous faisons ensuite usage d'un **invariant de boucle**, *i.e* d'une propriété vérifiée à chaque étape de la boucle principale de notre procédure.

Proposition IV.1. Soit L une liste de taille n . Alors pour tout $i \in \llbracket 0, n-1 \rrbracket$, la sous-liste $[L[0], \dots, L[i]]$ est triée.

Démonstration. Nous démontrons cet invariant par récurrence (finie) sur $i \in \llbracket 0, n-1 \rrbracket$.

- $i = 0$: trivial.
- Si on suppose la propriété vraie au rang $i \leq n-2$, alors on pose $x = L[i+1]$ et on distingue deux cas. Soit $L[i] \leq x$ et dans ce cas tout est déjà trié (on ne rentre pas dans la boucle `while`, soit la boucle `while` s'itérera jusqu'à atteindre un tel cas de figure. Plus précisément, au bout de j itérations de cette boucle, on aura trouvé un indice j tel que $L[j-1] \leq x$ et alors on sortira de la boucle en insérant x entre les éléments d'indices j et $j-1$ de L .

□

Corollaire IV.1.a. La procédure `triInsertion` est correcte.

Démonstration. Il suffit d'appliquer l'invariant de boucle *supra* au cas $i = n$. □

c) Complexité

Lorsque l'on calcule la complexité d'un algorithme de tri, on cherche en général à estimer le **nombre de comparaisons** effectué dans le pire cas possible, le meilleur cas possible et en moyenne. Nous exprimerons cette quantité en fonction de la taille n de la liste à trier.

- ▷ Dans le **meilleur cas**, la liste L est déjà triée et on ne rentre pas dans la boucle `while`. La complexité est alors clairement en $\mathcal{O}(n)$.
- ▷ Le **pire cas** correspond au cas où l'on atteint l'amplitude maximale de la boucle `while` à chaque étape de la boucle `for` : il faut donc pour cela que la liste soit triée en sens inverse. Étant donné qu'à l'étape i de la boucle `for`, la

boucle `while` effectue au plus i comparaisons, la complexité est dans ce cas donnée par :

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

- ▷ La **complexité en moyenne** du tri par insertion est hors-programme en MP. On peut toutefois démontrer qu'elle est équivalente à $\frac{n^2}{4}$ (et donc quadratique).

✂ **Remarque IV.1.** La complexité spatiale du tri par insertion tel qu'implémenté ici est négligeable : on agit directement sur la liste initiale sans en créer de copie ni manipuler de variables intermédiaires de grande taille.

2. Tri rapide

a) Diviser pour mieux régner

Le tri rapide est un tri **récuratif** basé sur un approche de type *divide and conquer* (diviser pour mieux régner en français). On procède selon le principe suivant :

- ▷ si la liste est vide ou réduite à un élément, elle est triée ;
- ▷ sinon, on fixe un élément e de la liste et on l'en extrait. On trie ensuite récursivement la liste I des éléments de la liste de base inférieurs ou égaux à e et sa consœur S des éléments strictement supérieurs à e puis on renvoie $I \cup [e] \cup S$.

✂ **Exercice IV.1.** Trier suivant cette méthode la liste $[2, 5, 3, 1, 10, -1]$.

Une implémentation possible de ce tri en `python` est donné par la fonction suivante.

```
def quicksort(L):
    if len(L) <= 1:
        return L
    I, S = [], []
    e = L[0] #on pourrait prendre n'importe qui ici
    for x in L[1:]: #pour éviter e
        if x <= e:
            I.append(x)
        else:
            S.append(x)
    return quicksort(I) + [e] + quicksort(S)
```

✂ **Remarque IV.2.** Il s'agit cette fois-ci d'une **fonction** : on renvoie une copie triée de la liste initiale **sans la modifier**.

b) Correction et terminaison

Une fois de plus, démontrer la terminaison de notre algorithme de tri est assez rapide : il suffit de remarquer que la longueur maximale des arguments passés en paramètre à chaque "division" de la liste initiale constitue une suite strictement décroissante d'entiers naturels.

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

Démonstration. On le démontre par récurrence forte sur la taille n de la liste L passée en argument.

- Si $n \in \{0, 1\}$, tout va bien.
- Si on suppose la fonction correcte pour toute liste de taille inférieure ou égale à un certain $n \geq 1$, alors pour tout liste L de taille $n + 1$, les appels récursifs `quicksort(I)` et `quicksort(S)` renverront deux listes triées car I et S sont de taille au plus n . De fait, la liste `quicksort(I)+[e]+quicksort(S)` est triée par définition de I et S .

□

c) Complexité

Comme précédemment, nous cherchons à évaluer la complexité temporelle de la fonction `quicksort` dans trois cas distincts en fonction de la taille n de la liste à trier.

- ▷ Pour étudier le meilleur cas, simplifions nous la vie en supposant, sans perte excessive de généralité, que la taille n de notre liste est de la forme $n = 2^p$. Pour effectuer le moins de comparaisons possibles, il faudrait idéalement que notre pivot e soit tel que $\text{len}(I) = 2^{p-1}$; ceci nous donnerait, en notant u_p le nombre de comparaisons nécessaires au tri d'une liste de taille 2^p , la relation de récurrence suivante :

$$\forall p \geq 1, u_p = 2^p + 2u_{p-1}$$

car la construction de I et S requiert n comparaisons. On obtient donc, par sommation :

$$u_p = \mathcal{O}(p2^p)$$

ce qui entraîne que la complexité de la fonction `quicksort` dans le meilleur des cas est en $\mathcal{O}(n \ln(n))$.

- ▷ Dans le **pire cas**, une des deux listes I et S contient à chaque étape tous les éléments à l'exception du pivot. On obtient alors, en notant C_n le nombre de comparaisons recherché, la relation

$$\forall n \geq 1, C_n = n + C_{n-1}$$

et donc, comme pour le tri par insertion, $C_n = \mathcal{O}(n^2)$.

- ▷ On peut démontrer (ceci étant hors-programme), que la **complexité moyenne** de ce tri est en $\mathcal{O}(n \ln(n))$. Inventé en 1961, le tri rapide est encore de nos jours le tri par défaut de nombreux langages de programmation.

3. Tri fusion

a) Interclassement, tri fusion

Le tri fusion est, lui aussi, un tri **récurif** basé sur une approche "diviser pour régner". L'idée centrale est la suivante : étant donnée une liste, on la coupe en deux,

on trie récursivement chaque moitié puis on les interclasse en comparant successivement leurs éléments minimaux respectifs.

▣► **Exemple IV.1.** Pour interclasser $[1,6,7]$ et $[2,3,5]$, on compare leurs éléments minimaux : $1 \leq 2$ donc on ajoute à une nouvelle liste 1 avant d'itérer ce procédé, en comparant cette fois $[6,7]$ et $[2,3,5]$. *In fine*, notre nouvelle liste contiendra $[1,2,3,5,6,7]$.

Afin d'implémenter le tri fusion, il faut donc commencer par programmer cette étape, fondamentale, d'interclassement. Nous proposons le code python suivant à cet effet.

```
def intercl(L1,L2):
    i,j=0,0
    n1,n2=len(L1),len(L2)
    L=[]
    while i<n1 and j<n2:
        if L1[i]<=L2[j]:
            L.append(L1[i])
            i+=1
        else:
            L.append(L2[j])
            j+=1
    if i==n1 and j<n2:
        L+=L2[j:]
    elif i<n1 and j==n2:
        L+=L1[i:]
    return L
```

Une fois ceci fait, nous pouvons proposer la fonction suivante pour réaliser le tri fusion.

```
def triFusion(L):
    if len(L)<=1:
        return L
    m=len(L)//2
    return intercl(triFusion(L[:m]),triFusion(L[m:]))
```

▣◻ **Exercice IV.2.** Détailler les étapes du tri fusion de la liste $[2,-1,4,5,3,6,18,0]$.

b) Correction et terminaison

Nous laissons en exercice au lecteur les preuves de terminaison et correction des deux algorithmes présentés *supra*. Pour la fonction `intercl`, nous suggérons l'usage de l'invariant de boucle suivant : à chaque étape, la liste L est triée et composée d'éléments inférieurs à ceux non parcourus de L1 et L2. La correction de la fonction `triFusion` se démontre aisément par récurrence forte sur la longueur de la liste à trier.

c) Complexité

De façon similaire à ce qui a été fait pour le tri rapide, on démontre que la complexité dans le pire et le meilleur des cas du tri fusion est en $\mathcal{O}(n \ln(n))$, où

n est la longueur de la liste à triée. Il en découle que la complexité moyenne est également en $\mathcal{O}(n \ln(n))$.

✂ **Remarque IV.3.**

- ▷ La complexité de l'interclassement est en \mathcal{O} de la somme des longueurs des listes considérées.
- ▷ La complexité en espace du tri fusion est considérable, car nous créons de nombreuses copies de listes.

Chapitre V

Kit de survie en terre SQL

Ce qui suit a pour vocation de rappeler au bon souvenir du lecteurs quelques notions vues en MPSI et de lui offrir une référence rapide des fonctions du langage SQL (Structured Query Language) traditionnellement usitées aux concours.

1. De quoi cause-t-on ?

Le langage SQL permet la manipulation de **bases de données relationnelles**, *i.e* d'ensembles **structurés** de **données informatiques** tels que :

- ▷ les données soient enregistrées sur un **support permanent** ;
- ▷ les données ne présentent **pas de redondance** ;
- ▷ chaque objet stocké possède un **identifiant unique**.

Une telle structure sera supposée constituée d'un ensemble de tables (ou p -uplets).

▣► **Exemple V.1.** Nous nous référerons dans cet appendice à la combinaison des deux tables suivantes, dont on ne cite qu'un extrait ici.

Table élèves

id	nom	prénom	classe	option
1034	Arnold	Bernard	MP	informatique
1035	Dupont	Bérénice	PTSI	allemand
⋮	⋮	⋮	⋮	⋮

Table notes

id	valeur	élève	discipline	date
589	3	1034	chimie	03.02.2019
590	17	1035	SII	03.03.2020
⋮	⋮	⋮	⋮	⋮

Vocabulaire. On appelle :

- ▷ **attribut** le nom d'une colonne dans une table donnée ;
- ▷ **domaine** d'un attribut l'ensemble de ses valeurs possibles ;
- ▷ **clé** tout sous-ensemble d'attribut permettant d'identifier sans ambiguïté une ligne de la table. Si ce sous-ensemble est minimal, on parle de **clé primaire**.

2. Interrogation d'une base de données

a) Projection

La **projection** est une opération du langage SQL permettant le **choix d'une colonne**. La syntaxe en est la suivante.

```
SELECT attributs
FROM table
```

▣► **Exemple V.2.** L'instruction

```
SELECT id, nom, prénom
FROM élèves
```

renverra la liste des identifiants, noms et prénoms des élèves listés dans la table élèves.

✂ **Remarque V.1.**

- ▷ Il est possible de projeter l'intégralité des attributs de la table via l'instruction `SELECT *`.
- ▷ L'instruction `LIMIT` permet de sélectionner les résultats que l'on souhaite afficher. Par exemple, l'instruction :

```
SELECT id, nom, prénom
FROM élèves
LIMIT 0,30
```

n'affichera que les 31 premiers triplets rencontrés dans la table.

- ▷ Mathématiquement, si \mathcal{R} est la relation associée à une table de notre base de donnée et A_1, \dots, A_k les attributs que nous souhaitons sélectionner, le résultat de la projection sera noté :

$$\Pi_{A_1, \dots, A_k}(\mathcal{R}).$$

b) Sélection

La **sélection** est une opération SQL consistant à sélectionner (ah ah) des éléments de notre base de donnée en **fonction des valeurs de leurs attributs**. La syntaxe en est la suivante.

```
SELECT attributs
FROM table
WHERE conditions
```

▣► **Exemple V.3.** L'instruction

```
SELECT discipline
FROM notes
WHERE valeur = 20
```

listera toutes les disciplines dans lesquels au moins un élève a obtenu la note 20.

✂ **Remarque V.2.**

- ▷ On peut éliminer les doublons à l'affichage en utilisant l'instruction `SELECT DISTINCT`.
- ▷ Mathématiquement, si \mathcal{R} est la relation associée à notre base de donnée et \mathcal{F} l'ensemble des conditions souhaitées, la sélection se note $\sigma_{\mathcal{F}}(\mathcal{R})$. On peut évidemment combiner ceci avec une projection, en écrivant

$$\Pi_{A_1, \dots, A_k}(\sigma_{\mathcal{F}}(\mathcal{R})),$$

avec A_1, \dots, A_k des attributs.

c) Jointure symétrique

La **jointure symétrique** est une opération SQL permettant de "**lier**" deux bases de données à travers un attribut commun. La syntaxe en est la suivante.

```
SELECT table1.attributs, table2.attributs
FROM table1 JOIN table2 ON table1.lien = table2.lien
WHERE conditions
```

▣► **Exemple V.4.** L'instruction

```
SELECT DISTINCT élèves.nom, élèves.prénom,
           notes.discipline
FROM élèves JOIN notes ON élèves.id = notes.élève
WHERE notes.valeur = 20
```

renvoie les noms et prénoms des élèves ayant obtenu au moins une fois la note 20, en éliminant les doublons et en précisant la discipline où la note a été obtenue.

✎ **Exercice V.1.** Que renvoie l'instruction suivante ?

```
SELECT DISTINCT élèves.nom
FROM élèves JOIN notes ON élèves.id = notes.élève
WHERE élèves.nom NOT IN
  ( SELECT élèves.nom
    FROM élèves JOIN notes ON élèves.id = notes.élève
    WHERE notes.valeur < 15 )
```

☞ **Remarque V.3.** Mathématiquement, si \mathcal{R} et \mathcal{R}' sont deux relations correspondant à deux tables que T est une égalité d'attributs permettant la jointure, celle-ci est notée $\mathcal{R} \bowtie_T \mathcal{R}'$.

d) Autres fonctions du langage SQL

Nous listons ici quelques fonctions communes du langage SQL dont nous estimons la connaissance utile au lecteur.

- ▷ L'instruction `ORDER BY` permet de trier les résultats d'une projection/sélection selon la valeur d'un attribut. On peut lui ajouter `DESCENDING` pour trier dans l'ordre inverse.

▣► **Exemple V.5.** L'instruction

```
SELECT id, valeur, discipline
FROM notes
ORDER BY valeurs
```

listera les notes présentes dans la table notes, classées par ordre croissant.

- ▷ L'instruction `GROUP BY` permet de grouper les résultats d'une projection/sélection selon la valeur d'un attribut. Un seul retour sera effectué par "classe" de groupement.
- ▷ L'instruction `COUNT` permet de compter le nombre de résultats d'une requête.

▣► **Exemple V.6.** L'instruction

```
SELECT DISTINCT valeur, COUNT(valeur)
FROM notes
WHERE discipline = EPS
```

renverra les notes listées en EPS ainsi que le nombre de fois où elles ont été obtenues.

- ▷ Les fonctions `MAX`, `MIN`, `SUM`, `AVG` permettent de calculer respectivement le maximum, le minimum, la somme et la moyenne des résultats d'une requête.

▣► **Exemple V.7.** L'instruction

```
SELECT élèves.nom, AVG(notes.valeur)
FROM élèves JOIN notes ON élèves.id = notes.élève
GROUP BY notes.élève
```

renverra les noms des élèves de la base ainsi que la moyenne de leurs notes en toutes disciplines.

- ▷ Il est possible de donner un alias à une table via l'instruction `AS`. Ceci peut permettre de joindre une table à elle-même.

▣► **Exemple V.8.** L'instruction

```
SELECT t1.prénom, t2.prénom
FROM élèves AS t1 JOIN élèves AS T2
      ON t1.id=t2.id
WHERE t1.nom=t2.nom
```

renvoie la liste des prénoms d'élèves ayant le même nom de famille.

Aide-mémoire de complexité

Nous nous préoccupons principalement ici de complexité temporelle; il s'agit souvent en pratique de compter le nombre d'opérations élémentaires (souvent multiplications, et à défaut additions) demandées par un appel à l'algorithme étudié. On donnera rarement une valeur exacte pour ce nombre : un \mathcal{O} est amplement suffisant.

La complexité spatiale (occupation de la mémoire vive) sera généralement laissée de côté. Il est toutefois intéressant de garder en têtes les quelques cas où cela a une importance listés dans ce cours.

Le tableau qui suit a pour but d'aider le lecteur à visualiser les différentes complexités temporelles classiques, leur nom et leur degré de coolitude. Il ne saurait se substituer à un véritable cours sur la complexité. Le paramètre n présenté est ici la **taille** de l'objet étudié : pour une liste ou un tableau, il s'agira du nombre d'éléments. Pour un entier ou un flottant, il s'agira de son nombre de chiffres.

Complexité	Nom	Raisnable ?
$\mathcal{O}(1)$	temps constant	cas idéal
$\mathcal{O}(\ln(n))$	complexité logarithmique	très
$\mathcal{O}(n)$	complexité linéaire	oui
$\mathcal{O}(n \ln(n))$	complexité semi-logarithmique	oui
$\mathcal{O}(n^2)$	complexité quadratique	oui
$\mathcal{O}(n^k) \ (k \geq 1)$	complexité polynomiale	généralement
$\mathcal{O}(e^n)$	complexité exponentielle	non
$\mathcal{O}(n!)$	complexité factorielle	Cthulhu fhtagn