

Informatique MPSI / MP : L'essentiel

Louis Alvin

2 juin 2021

1 Représentation des nombres

Entiers (\mathbb{N}) sur n bits : 2^n valeurs : de 0 à $2^n - 1$.

Pour convertir un nombre en binaire : méthode des divisions euclidiennes successives par 2.

Entiers relatifs (\mathbb{Z}) Méthode du complément à 2.

Si $x > 0$, on le représente sur $n - 1$ bits. Si $x < 0$:

- 1) décomposition en binaire de $2^n - x$
 - 2) écriture binaire de $-x \rightarrow$ inverser les bits \rightarrow ajouter 1.
- En complément à 2, on peut représenter $\llbracket -2^{n-1} ; 2^{n-1} - 1 \rrbracket$

Réels (\mathbb{R}) : $y = (1, b_1 b_2 \dots b_k)$; norme IEEE-754 sur 64 bits.

- 1 bit de signe s
- exposant p sur 11 bits, représenté par $p + 1023$
- mantisse m sur 52 bits

2 Bases de données relationnelles – SQL

2.1 Opérateurs de base

Projection $\prod_{c_1, \dots, c_p}(t)$: `SELECT [DISTINCT] client FROM achats`

Sélection $\sigma_c(t)$: `SELECT * FROM achats WHERE prix > 40`

Conditions Opérateurs de comparaisons : $=, !=, >, \geq, \dots, IS [NOT] NULL$.

Opérateurs booléens : AND, OR, NOT.

Trier `SELECT ... FROM ... ORDER BY ... ASC/DESC`

Renommage $\rho_{a \rightarrow b}(t)$:

```
SELECT client, prix - prix*remise/100 AS prix_reduit  
FROM achats WHERE prix_reduit < 55
```

Opérateurs ensemblistes UNION, INTERSECT (sauf pour MySQL); EXCEPT (pour SQLite, MINUS sur MySQL).

2.2 Opérateurs plus complexes

Fonctions d'agrégation `SELECT MAX(prix) FROM achats`

Fonctions disponibles : MAX, MIN, SUM, AVG, COUNT

(COUNT(c) \rightarrow nb colonnes non NULL ; COUNT(*) \rightarrow nb total de colonnes)

Requêtes agrégatives GROUP BY et HAVING

N.B. : WHERE \rightarrow avant les groupes ; HAVING \rightarrow après les groupes

Produit cartésien ; Jointures Dans l'algèbre relationnelle, la jointure symétrique de deux tables t et t' selon les attributs a et a' , est notée : $t[a = a']t'$ ou $t \bowtie_{a=a'} t'$ et est définie par : $t \bowtie_{a=a'} t' = \sigma_{a=a'}(t \times t')$.

En SQL : `SELECT * FROM batiments JOIN chambres`

`ON batiments.nom = chambres.batiment`

LIMIT et OFFSET (tout à la fin d'une requête, généralement après ORDER BY)

LIMIT → renvoyer un nombre de lignes limité

OFFSET → effectue un décalage : nombre de lignes à passer avant de renvoyer le résultat

3 Algorithmes numériques

3.1 Recherche dichotomique dans une liste triée

```
def recherche_dicho(x,L):
    i = 0
    j = len(L)-1
    while i <= j:
        p = (i+j)//2
        if L[p] == x:
            return True
        elif L[p] > x:
            j = p - 1
        else:
            i = p + 1
    return False
```

3.2 Calcul d'intégrale (méthode des rectangles)

On utilise la méthode des rectangles (gauches), dont la formule est :

$$I_{a,b}^R(f) = \frac{b-a}{n} \times \sum_{i=0}^{n-1} f(x_i)$$

```
def rectangles(f,a,b,n):
    integrale = 0
    h = (b-a)/n
    x = a
    for i in range(n):
        integrale += f(x)
        x = x+h
    return h * integrale
```

Remarque : méthode des trapèzes : on approxime $\int_a^b f(x)dx$ par $\sum_{i=0}^{n-1} h \times \frac{f(x_i)+f(x_{i+1})}{2}$.

3.3 Résolution d'équation par recherche dichotomique

La fonction suivante renvoie une valeur approchée d'un zéro de f sur $[a, b]$ avec une précision `epsilon`. $f(a)$ et $f(b)$ de signes opposés, $a < b$, $\varepsilon > 0$.

```
def dichotomie(f, a, b, epsilon):
    u,v = a,b
    while v-u >= epsilon:
        milieu = (u+v)/2
        if f(u) * f(milieu) < 0:
            v = milieu
        else:
            u = milieu
    return u
```

3.4 Résolution d'équation par la méthode de Newton

Conditions : f doit être de classe C^1 sur $[a, b]$ et $f(a)$ et $f(b)$ doivent être de signe opposé.

La méthode de Newton consiste à itérer la suite x_n définie par :

$$\begin{cases} x_0 \in [a, b] \\ \forall n \in \mathbb{N}, x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \end{cases}$$

```

def newton_residu(f, fprime, x0, epsilon):
    x = x0
    while (abs(f(x)) > epsilon):
        x = x - f(x)/fprime(x)
    return x

def newton_increment(f, fprime, x0, epsilon):
    x, y = x0, x0 - f(x0)/fprime(x0)
    while (abs(y - x) > epsilon):
        x, y = y, y - f(y)/fprime(y)
    return x

```

3.5 Résolution d'équation différentielle par la méthode d'Euler

Ordre 1 On considère une équation différentielle de la forme $\begin{cases} y' = f(t, y) \\ y(t_0) = y_0 \end{cases}$ (condition initiale) sur l'intervalle $[a, b]$ (en général $t_0 = a$).

On souhaite approximer la fonction y en un certain nombre d'instants répartis sur $[a, b]$.

On note y_i l'approximation de la fonction y à l'instant t_i . On détermine les y_i de proche en proche :

$$y(t_{i+1}) = y(t_i + h) \underset{\text{(D.L.)}}{=} y(t_i) + h y'(t_i) + o(h) \underset{\text{(y sol.)}}{=} y_i + h f(t_i, y(t_i)) + o(h) \approx y_i + h f(t_i, y_i)$$

Au final, on a donc : $\begin{cases} y_{i+1} = y_i + h \times f(t_i, y_i) \\ t_{i+1} = t_i + h \end{cases}$ avec $\begin{cases} y_0 \in \mathbb{R} \\ t_0 = a \end{cases}$

```

def euler(f, a, b, y0, n):
    h = (b-a)/n
    t, y = a, y0
    T, Y = [a], [y0]
    for i in range(n):
        y = y + h * f(t,y)
        Y.append(y)
        t = t + h
        T.append(t)
    return T, Y

```

Ordre 2 Pour résoudre numériquement un système différentiel contenant n équations, on le voit comme une seule ED de la forme $Y'(t) = F(t, Y(t))$ où $t \rightarrow Y(t) \in \mathbb{R}^n$ est **une fonction vectorielle**.

Conséquence : dans la méthode d'Euler, la relation de récurrence vérifiée par les (y_i) est la même.

$$\text{Euler ordre 1} \quad y_{i+1} = y_i + h \times f(t_i, y_i) \in \mathbb{R}$$

$$\text{Euler systèmes diff.} \quad Y_{i+1} = Y_i + h \times F(t_i, Y_i) \in \mathbb{R}^n$$

Implémentation Python $\text{y} = \text{y} + \text{h} * \text{f}(\text{t}, \text{y})$

Attention : pour l'implémentation, on n'utilise pas des listes mais des tableaux Numpy. Par exemple :

```

def F(t,Y):
    u,v = Y # ou encore : u,v = Y[0],Y[1]
    x = 3*u + 4*v + t
    y = 4 * np.cos(u) - 3*t
    return np.array([x,y])

```

On peut alors utiliser la fonction **euler** telle quelle : `LT,LY = euler(F,a,b,Y0,n)`.

Ordre supérieur On considère le vecteur $Y(t) = \begin{pmatrix} y(t) \\ y'(t) \end{pmatrix}$.

Alors $Y'(t) = \begin{pmatrix} y'(t) \\ y''(t) \end{pmatrix} = \begin{pmatrix} y'(t) \\ 2y'(t) + \sin(y(t)) + t^2 \end{pmatrix} = F(t, Y(t))$ ne dépend que de t , $y(t)$ et $y'(t)$.

En Python :

```

def F(t,Y):
    # Y = (y,y')
    return np.array([Y[1], 2*Y[1] + np.sin(Y[0]) + t**2])

```

Cette méthode se généralise aux ED d'ordre $p \geq 3$ et aux systèmes différentiels d'ordre $p \geq 2$.

3.6 Résolution de système linéaire

On considère un système linéaire qu'on peut écrire matriciellement sous la forme :

$$\begin{pmatrix} a_{0,0} & \dots & a_{0,n-1} \\ \vdots & & \vdots \\ a_{n-1,0} & \dots & a_{n-1,n-1} \end{pmatrix} \begin{pmatrix} x_0 \\ \vdots \\ x_{n-1} \end{pmatrix} = \begin{pmatrix} b_0 \\ \vdots \\ b_{n-1} \end{pmatrix} \iff AX = B, \quad A \in \mathcal{M}_n(\mathbb{R}), X \in \mathbb{R}^n \text{ (inconnue)}, B \in \mathbb{R}^n.$$

On suppose que $A \in GL_n(\mathbb{R})$. Donc $\exists! X \in \mathbb{R}^n, AX = B$.

Les matrices sont représentées par des tableaux Numpy à deux dimensions. On veillera à distinguer les opérations qui copient un tableau de celles qui le modifient directement → voir cours : *Slices and views – Coupes et vues*.

Exemples : si $A = \begin{pmatrix} 1 & 4 & -3 \\ -4 & 0 & 2 \end{pmatrix}$, $b = A[1]$ contient $(-4 \ 0 \ 2)$; $c = A[:,2]$ contient $\begin{pmatrix} -3 \\ 2 \end{pmatrix}$.

Une instruction telle que $A[1,2] = 7$ modifie aussi b (qui vaut alors $(-4 \ 0 \ 7)$) et c : ce sont des vues.

On utilise cela lors de l'échange de deux lignes, pour éviter de copier le tableau entier.

Principe général Étape 1 : descente : on met le système sous forme triangulaire supérieure.

Étape 2 : remontée : on résout le système triangulaire.

Pour la formalisation, voir cours (Chap 15).

```
def echange(A,i,j):
    tampon = A[i].copy()
    A[i] = A[j]
    A[j] = tampon

def recherche_pivot(A,j): # naïf
    i = j
    while A[i,j] == 0:
        i = i + 1
    return i

def elimination(A,b,j):
    i = recherche_pivot(A,j)
    echange(A,i,j)
    echange(b,i,j)
    for k in range(j+1, A.shape[0]):
        b[k] = b[k] - (A[k,j]/A[j,j]) * b[j]
        A[k] = A[k] - (A[k,j]/A[j,j]) * A[j]

def descente(A,b):
    for j in range(A.shape[1]-1):
        elimination(A,b,j)

def remonte(A,b):
    n = A.shape[0]
    X = np.zeros(n)
    for k in range(n-1, -1, -1):
        somme = 0
        for l in range(k+1, n):
            somme = somme + A[k,l]*X[l]
        X[k] = (b[k]-somme) / A[k,k]
    return X

def resolution(A,b):
    A_copie = A.copy()
    b_copie = b.copy()
    descente(A_copie, b_copie)
    return remonte(A_copie, b_copie)
```

4 Algorithmes de tri

4.1 Tri insertion

On insère $L[i]$ à la bonne place dans la portion $L[:i]$ déjà triée, pour $i \in [1; \text{len}(L) - 1]$.

Deux méthodes : soit on effectue des échanges à gauche tant que l'élément à insérer est inférieur à son voisin de gauche, soit on décale les éléments à déplacer avant d'insérer le nouvel élément.

```
def tri_insertion(L):
    for i in range(1, len(L)):
        x = L[i]
        j = i - 1 # indice de l'élément à comparer avec x
        while j >= 0 and x < L[j]:
            L[j+1] = L[j] # recopie à droite
            j = j - 1
        L[j+1] = x # insertion de x à la bonne place
```

Pas de `return` L : la liste est triée *en place*.

Complexité Dans le meilleur cas, si la liste est déjà triée, la complexité de ce tri est en $\mathcal{O}(n)$.

Dans le pire cas, L est triée dans l'ordre décroissant. On a alors une complexité en $\mathcal{O}(n^2)$.

Complexité moyenne en $\mathcal{O}(n^2)$ également.

4.2 Tri fusion

On coupe la liste en deux portions de taille presque égales qu'on trie récursivement, puis on fusionne en plaçant à chaque étape le plus petit des éléments en tête de liste à fusionner à la suite des éléments déjà extraits.

Exemple : $[2, 7, 1, 4, 3, 6, 5, 8] \rightarrow [2, 7, 1, 4] \quad [1, 2, 4, 7] \rightarrow [1, 2, 3, 4, 5, 6, 7, 8]$

```
def fusion(L1,L2):
    L = []
    while L1 != [] and L2 != []:
        if L1[0] > L2[0]:
            L.append(L2.pop(0))
        else:
            L.append(L1.pop(0))
    return L + L1 + L2

def tri_fusion(L):
    if len(L) <= 1:
        return L
    L1 = tri_fusion(L[:len(L)//2]) # // et pas /
    L2 = tri_fusion(L[len(L)//2:])
    return fusion(L1,L2)
```

Fusion : On peut se passer des `pop()` en gardant en mémoire deux indices correspondant aux éléments de L1 et L2 à comparer. À la sortie du `while`, une seule des deux listes L1 et L2 est vide et l'autre contient des éléments supérieurs à ceux de L.

Remarque : Ce n'est pas un tri en place (à cause des slicings).

Complexité $T(n) = \mathcal{O}(n \log n)$ dans tous les cas (meilleur, pire, moyen).

4.3 Tri rapide

Aussi appelé *Tri partition*, ou *Quicksort* en anglais.

Principe On choisit un pivot $p \in L$.

La partition consiste à mettre la liste sous la forme : $\boxed{\leq p} \boxed{p} \boxed{> p}$. Puis on trie récursivement les deux parties $\leq p$ et $> p$.

Comment partitionner ? Schéma de Lomuto : à chaque étape, L est de la forme :

deb	i	j	fin
↓	↓	↓	↓
p	$\leq p$	$> p$???

Initialement, $\boxed{\leq p}$ et $\boxed{> p}$ sont vides, donc $i = j = \text{deb} + 1$.

Au tour de boucle j, si $L[j] > p$, il est à la bonne place et on ne fait rien ; si $L[j] \leq p$, on échange $L[j]$ et $L[i]$.

```

def partition(L,deb,fin):
    i = deb + 1
    p = L[deb]
    for j in range(deb+1, fin+1):
        if L[j] <= p:
            L[i],L[j] = L[j],L[i]
            i = i + 1
    L[deb],L[i-1] = L[i-1],L[deb]
    return i - 1 # /!\

def tri_partition(L,deb,fin):
    if deb < fin:
        k = partition(L,deb,fin)
        tri_partition(L,deb,k-1)
        tri_partition(L,k+1,fin)

```

C'est un tri en place. Pour trier une liste en entier, on exécute `tri_partition(L, 0, len(L)-1)`.

Complexité $\mathcal{O}(n^2)$ dans le pire cas. $\mathcal{O}(n \log n)$ dans le meilleur.