Informatique - Algorithmes

1. Calculs itératifs

1.1. Calcul du n-ième terme d'une suite récurrente $u_{n+1} = f(u_n)$

ici $u_0 = 2$ et $\forall n \in \mathbb{N},\ u_{n+1} = \frac{1}{2}\left(u_n + \frac{2}{u_n}\right)$

```
def suite(n):
    u = 2
    for i in range(n):
        u = (u + 2/u)/2
    return u
```

1.2. Calcul d'une somme, d'un produit

Pour $\sum_{k=1}^{n} \frac{1}{k}$ et et pour $n! = \prod_{k=1}^{n} k$

```
def somme(n):
    S=0
    for i in range(1,n+1):
        S = S + 1/i    # ou S += 1/i
    return S
```

```
def produit(n):
    P = 1
    for i in range(1,n+1):
        P = P*i    # ou P *= i
    return P
```

2. Algorithmes de recherche

2.1. Recherche d'un élément x dans une liste L non triée

```
def recherche1(L,x) :
    i = 0        # on cherche l'indice de
                 # la 1ère occurences de x
    while i< len(L) and L[i] != x :
        i += 1
    return i
```

```
def Recherche2(L,x) :
    indices = []  # liste des occurences
    for i in  range(len(L)) :
        if L[i] == x :
            indices.append(i)
    return indices   # vide si non trouvé
```

2.2. Recherche du maximum d'une liste L

```
def rechercheMaxi(L) :
    M = L[0]    # le premier est le plus grand
    for val in L :
        if val > M : # si l'élément courant
            M = val  # est plus grand
    return M
```

2.3. Recherche dichotomique de la première occurence d'un élément val dans une liste L triée

On maintient l'invariant de boucle **g < val ⩽ d**.

```
def rechercheDichotomique(val,L) :
    g = -1    # on cherche la 1ère
    d = len(L)-1    # occurence de val
    while g+1 < d :    # arrêt sur g+1 == d
        m = (g+d)//2
        if L[m] < val : # on garantit
            g = m    # l'invariant
        else :    # L[g] < val <= L[d]
            d = m
    if L[d] == val:    # d'après l'invariant
        return d    # val est en d
    else:    # sinon
        return None    # val n'est pas dans L
```

2.4. Recherche d'un motif (mot) M dans une chaîne de caractères c

```
def rechercheMotif(M,c) :
    """ recherche le motif M dans la chaine
        de caractères c"""
    for d in range(len(c) - len(M) +1) :
        k = 0
        while k<len(M)-1 and M[k]==c[d+k] :
            k = k+1
        if  M[k] == c[d+k] :  # si on a trouvé
            return d    # le motif en entier
    return None
```

3. Calculs sur une liste

3.1. Calcul de la moyenne ou de la variance d'une liste L de données numériques

```
def moyenne(L) :
    S=0
    for element in L :
        S = S + element
    return S/len(L)
```

```
def variance(L) :
    S1 = 0
    S2 = 0
    for i in range(len(L)) :
        S1 = S1 + L[i]
        S2 = S2 + L[i]**2
    V = S2/len(L) - (S1/len(L))**2
    return V
```

4. Intégration numérique

4.1. Méthode des rectangles

```
def rectangles(f,a,b,n) :
    S = 0    # point à gauche
    pas = (b-a)/n   # n rectangles
    for k in range(n) :
        S += f(a+k*pas)
    return S*pas
```

4.2. Méthode des trapèzes

```
def trapezes(f,a,b,n) :
    S = (f(a) + f(b))/2
    pas = (b-a)/n  # n trapèzes
    for k in range(1,n) :    # 1 à n-1
        S += f(a+k*pas)
    return S*pas
```

4.3. En utilisant numpy

```
def trapezes(f,a,b,n) :
```

```
    X = np.linspace(a,b,n+1) # n+1 points
    S = 0.0
    for k in range(n) :
        S += (X[k+1]-X[k])*( f(X[k+1]) + f(X[k]) )/2
    return S
```

5. Résolution d'une équation non linéaire

5.1. Résolution de $f(x) = 0$ par dichotomie

```
def dichotomie(f,a,b,precis) :
    while abs(a -b)> precis :
        Xmilieu = (a + b)/2
        if f(a)*f(Xmilieu)<0 : # zéro entre
            b = Xmilieu    # a et Xmilieu
        else :
            a = Xmilieu
    return Xmilieu
```

5.2. Méthode de Newton

On s'arrête quand $|x_{n+1} - x_n| \leqslant \varepsilon$:

```
def zeroNewton(f,fprime,x0,epsilon) :
    x1 = x0 + 2*epsilon   # pour démarrer
    while abs(x1-x0) > epsilon :
        x1 = x0
        x0 = x1 - f(x1)/fprime(x1)
    return x0
```

6. Résolution d'une équation différentielle

6.1. Équation différentielle d'ordre 1

Résolution approchée de : $2t^2y'(t) - \dfrac{\cos(y)}{1+t^2} = 1$

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

def deriv(y,t):
    return (1+np.cos(y)/(1+t**2))/(2*t**2)

def EulerED1(f,y0,t):
    n=len(t)
    y = np.zeros(n) # tableau même taille
    y[0] = y0 #  condition initiale
    for k in range(n-1): # n-1 valeurs hors 0
        y[k+1] = y[k] + (t[k+1]-t[k])*f(y[k],t[k])
    return y

t = np.linspace(0,np.pi,100)
y = EulerED1(deriv,1.3,t)
plt.plot(t,y)
plt.show()
```

6.2. Équation différentielle d'ordre 2

Par exemple, pour une équation différentielle de la forme $y'' + a(t)y' + b(t)y = c(t)$, on aura $y'' = c(t) - a(t)y' - b(t)y$.

On pose $Y(t) = [y(t), y'(t)]$ et l'équation s'écrit $Y'(t) = F(Y,t)$.

```
def F(Y,t): # avec la convention Y=[y,y']
   dY = [Y[1],  c(t) - a(t)*Y[1] - b(t) *Y[0]]
   return np.array(dY)


def EulerEDv(F,Y0,t):
    n = len(t)
    Y = np.zeros((n,2)) # 2 coordonnées pour Y[k]
    Y[0] = Y0 #  condition initiale
    for k in range(n-1):
        Y[k+1] = Y[k] + (t[k+1]-t[k])* F(Y[k],t[k])
    return Y

# Y[:,0] contient les valeurs de u
# et Y[:,1] celles de v

t = np.linspace(tdebut,tfin,N)
# condition initiale y(0)=y0 et y'(0)=yy0
Y0 = np.array([y0,yy0])
solu = EulerEDv(F,Y0,t): # Euler vectorielle

plt.plot(t,solu[:,0])
#  y dans solu[:,0] et y' dans solu[:,1]
plt.show()
```

6.3. Équation différentielle d'ordre 2 avec scipy

$y''(t) = -2\lambda y'(t) - \omega_0^2 y(t)$ avec $y(0) = 2,5$ et $y'(0) = 0$

```
from scipy.integrate import odeint
import numpy as np
import matplotlib.pyplot as plt

def func(Y,t):
    return [Y[0],2 *lam*Y[0] - w0**2*Y[1]]

lam = 1.0 ; w0 = 3.0
t = np.linspace(0,10,100)
Y = odeint(func,[2.5,0.0],t)
plt.plot(t,Y[:, 0],label="y")
plt.plot(t,Y[:, 1],label="y'")
plt.show()
```

7. Matrices

7.1. Opérations élémentaires

```
def transposition(m,i,j):
    p = len(m[0])
    for k in range(p):
       m[j,k], m[i,k] = m[j,k],m[i,k]
    # ou plus simplement
    # m[i], m[j] = m[j],m[i].copy

def transvection(m,i,alpha,j):
    p = m.shape[1]
    for k in range(p):
        m[i,k]=m[i,k]+alpha*m[j,k]
    # ou plus simplement
    # m[i] = m[i] + alpha * m[j]

def dilatation(m,i,mu):
    n,p=m.shape
    for k in range(p):
        m[i,k] = mu*m[i,k]
    # ou plus simplement
    # m[i] = mu * m[i]
```

7.2. Recherche partielle du pivot

On recherche le coefficient le plus grand en valeur absolue dans la colonne du pivot sous la diagonale.

```
import numpy as np

def recherchePivot(m,k):
    n = m.shape[0] # nb de lignes
```

```
    jmax=k
    for j in range(k+1,n):
        if abs(m[j,k])>abs(m[jmax,k]):
            jmax = j
    return jmax
```

7.3. Résolution d'un système

On suppose que le système est de rang maximal.

On résout $aX = b$ et on note $m = (a|b)$ la matrice augmentée du système.

```
m = np.hstack((a,b))
```

```
def GaussJordan(m):
    """ m est la matrice augmentée du système """
    n,p = m.shape # on doit avoir p == n+1
    for k in range(n):
        # on cherche le meilleur pivot
        lpivot = rechPivot(m,k)
        transposition(m,lpivot,k)
        # on normalise le pivot
        dilatation(m,1/m[k,k],k)
        # on élimine au-dessus et au-dessous
        for j in range(n):
            # sauf sur la ligne courante
            if j != k:
                transvection(m,j,-m[j,k]/m[k,k],k)
    # la dernière colonne contient les solutions
    return m[:,n]
```

8. Tracé de fonctions

On utilise

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

8.1. Tracer la courbe d'une fonction

Si f est une fonction usuelle :

```
def traceCourbe(f,a,b) :
    x = np.linspace(a,b,num=200)
    y = f(x)
    plt.plot(x,y)
    plt.show()
```

Si f est une fonction non usuelle :

```
def traceCourbe(f,a,b) :
    x = np.linspace(a,b,num=200)
    vfonc = np.vectorize(f)
    y = vfonc(x)
    plt.plot(x,y)
    plt.show()
```

Pour les commandes artistiques :

```
x = np.linspace(a,b,num=200)
y1 = f(x)
y2 = g(x)
plt.plot(x,y1,"ro", label="courbe 1 ronds rouges")
plt.plot(x,y2,"b-", label="courbe 2 continue bleue")
plt.legend()
plt.xlabel(" x en m")
plt.ylabel(" y en m/s")
plt.title("titre du ...")
plt.grid()
plt.axis("equal")
plt.show()
```

ou

```
plt.legend( ( "courbe 1", "courbe 2") )
```

9. Lecture et écriture dans un fichier

9.1. Lire une valeur par ligne

Lecture d'une valeur par ligne dans le fichier **nom='MonFichier.txt'**

```
def lecture1(nom) :
    données = [ ]
    f = open(nom, 'rt')
    for ligne in f:
        données.append( float(ligne) )
    f.close()
    return données
```

9.2. Lire plusieurs valeurs par ligne

Lecture de plusieurs valeurs par ligne dans le fichier **nom='MonFichier.txt'** : temps et tension séparées par un espace

```
def lecture2(nom) :
    temps=[ ] ; tension =[ ]
    f = open(nom, 'r')
    # si besoin, on saute une ligne
    # li = f.readline()  #lire une seule ligne
    for ligne in f:
        L = ligne.split(' ')
        temps.append(float(L[0]))
        tension.append(float(L[1]))
    f.close()
    return (temps, tension)
```

9.3. Lecture de toutes les lignes

Lecture du texte depuis le fichier **nom='MonFichier.txt'**

```
f = open(nom,'rt')
liste_lignes = f.readlines()   # lit tout
f.close()
```

9.4. Lecture d'un texte depuis un fichier texte

Lecture du texte depuis le fichier **nom='MonFichier.txt'**

```
def lireTout(nom) :
    f = open(nom,'rt')
    chaine = f.read()       # lit tout
    f.close()
    return chaine
```

9.5. Ecriture dans un fichier texte

Création du fichier **nom = 'MonFichier.txt'** et écriture de la chaine de caractères **chaine** dans ce fichier

Le caractère **\n** marque la fin d'une ligne.

```
def EcrireTout(nom, chaine) :
    f = open(nom,'w')
    f.write(chaine)
    f.close()
    return None
```

10. Tri par insertion

10.1. Avec une fonction insertion

```
def insere(L,u):
    """ insère u à sa place dans L triée """
    L.append(u)
    k = len(L) - 1    # dernier indice
    while k>0 and L[k-1]>u :
```

```
        L[k] = L[k-1]  # on décale les éléments
        k = k-1
    L[k] = u # on insère u

def triInsertion(L):
    """ trie par insertion la liste L """
    n = len(L)
    nL =  L[:1] # nouvelle liste triée
    # variante : nL = [L[0]]
    for i in range(1,n):
        insere(nL, L[i])
    return nL
```

10.2. Tri insertion en place

```
def triInsertionEnPlace(L):
    """ trie en place par insertion la liste L """
    for i in range(1,len(L)):
        u = L[i]  # élément à insérer au début
        k = i    # on remonte à partir de i
        while k>0 and L[k-1]>u:
            L[k] = L[k-1]  # on décale les
            k = k-1    # éléments plus grands
        L[k] = u    # on insère l'élément u
```

11. Tri rapide

11.1. Avec une fonction de partition en place

```
def partition(L,a,b):
    """ partitionne la tranche L[a:b] """
    pivot = L[(a+b)//2]   # ou L[a] ou L[b] ou ...
    # on place le pivot en 1ère position
    L[a], L[(a+b)//2] = L[(a+b)//2], L[a]
    i = a+1
    for j in range(a+1, b):
        if L[j] <= pivot :
            L[i],  L[j] = L[j], L[i]
            i = i +1
    i = i-1
    L[i],  L[a] = L[a], L[i] # on replace le pivot
    return i    # on renvoie la position du pivot

def tri_rapide(L, a, a):
    if b <= a + 1: #  1 élément ou moins
        return None
    else:
        i = partition(L, a, b)
        tri_rapide(L, a, i) # on laisse le pivot
        tri_rapide(L, i+1, b)
        return None
```

12. Tri fusion

12.1. Avec inflation de l'utilisation de la mémoire

```
def fusion1(L1,L2):
    i1 = 0
    i2 = 0
    L=[]
    for k in range(0,len(L1) + len(L2)):
        if i2 >= len(L2) or
              (i1 < len(L1) and L1[i1] <= L2[i2]) :
            L.append(L1[i1])
            i1 += 1
        else:
            L.append(L2[i2])
            i2 += 1
    return L

def tri_fusion1(L):
    n = len(L)
    if n <=1:
        return L
    else:
        L1 = tri_fusion1(L[0:n//2])
        L2 = tri_fusion1(L[n//2:])
        return fusion1(L1,L2)
```

12.2. Version sans inflation de l'utilisation de la mémoire

```
def fusion(L,a,m,b):
    """ fusionne dans L les
        parties L[a:m] et L[m:b] """
    i1 = 0
    i2 = 0
    L1 = L[a:m]
    L2 = L[m:b]
    for k in range(a,b):
        if i2 >= len(L2) or
              (i1 < len(L1) and L1[i1] <= L2[i2]) :
            L[k] = L1[i1]
            i1 += 1
        else:
            L[k] = L2[i2]
            i2 += 1

def tri_fusion(L,a,b):
    if b-a <=1:
        return None
    else:
        tri_fusion(L,a,(a+b)//2)
        tri_fusion(L,(a+b)//2,b)
        fusion(L,a,(a+b)//2,b)
```

SELECT

La difficulté principale des requêtes **SELECT** est que l'ordre dans lequel sont données les commandes n'est pas tout à fait l'ordre dans lequel elles sont " logiquement" exécutées [1].

Sans agrégation

```
SELECT [DISTINCT] quoi
FROM ...
WHERE ...
ORDER BY ...
```

Ordre "logique" des manipulations :

- **FROM** : on accède à des tuples, éventuellement obtenus par un **JOIN** entre des tables et des sous-requêtes **(SELECT ...)**
- **WHERE** : on sélectionne ceux qui nous intéressent
- **quoi** : on dit quelles *valeurs* on veut afficher à propos de chaque tuple
- **ORDER BY** : les tuples étant classés suivant certaines *valeurs* croissantes (ou décroissantes **ORDER BY ... DESC**)

tuple : une ligne de données (une personne, une mesure, une ligne de production, etc.)

valeur associée à un tuple :
- un champ / attribut (une colonne) ;
- un calcul à partir d'autres *valeurs*
- le résultat d'une sous-requête s'appuyant sur d'autres *valeurs* et renvoyant une seule donnée

Avec agrégation

COUNT(...), **MAX(...)**, **MIN(...)**, **AVG(...)**, **SUM(...)**, **GROUP BY ...**

```
SELECT quoi
FROM ...
WHERE ...
GROUP BY ...
HAVING ...
ORDER BY ...
```

Ordre "logique" des manipulations :

- **FROM** : on accède à des tuples, éventuellement obtenus par un **JOIN** entre des tables et des sous-requêtes **(SELECT ...)**
- **WHERE** : on sélectionne ceux qui nous intéressent
- **GROUP BY** : on regroupe les tuples partageant une même *valeur* ; on obtient des groupes, les tuples ne jouent plus de rôle
- pas de **GROUP BY** : un seul groupe regroupant la totalité des tuples
- **HAVING** : en ne gardant que certains groupes pour lesquels certaines *valeurs* vérifient certaines propriétés
- **quoi** : on dit quelles *valeurs* on veut afficher à propos de chaque groupe
- **ORDER BY** : les groupes étant classés suivant certaines *valeurs* croissantes (ou décroissantes **ORDER BY ... DESC**)

valeur associée à un groupe :
- un champ / attribut, si dans chaque groupe, tous les tuples ont la même valeur dans ce champ ;
- une fonction d'agrégation appliquée au groupe ;
- un calcul à partir d'autres *valeurs*
- le résultat d'une sous-requête s'appuyant sur d'autres *valeurs* et renvoyant une seule donnée

COUNT(*) : compte tous les tuples du groupe ; syntaxe unique propre à **COUNT**, car aucune des autres fonctions d'agrégation n'a de sens sans préciser *de quoi* on fait la moyenne, le maximum, etc.

Calculs de valeurs : additions, multiplications, **COS(...)**, etc.

Pour **WHERE** ou **HAVING** :

BETWEEN, $=$, $\leqslant$, etc.

→**HAVING AVG(salaire) BETWEEN 1200 AND 1401**

IN (...) : dans un liste ou dans le résultat d'une sous-requête qui renvoie une colonne

→**WHERE prenom IN ('Bertrand', 'Jules')**

→**WHERE conge.date IN (SELECT date_naissance FROM ...)**

1. sans compter qu'en général, le moteur SQL suit sa propre logique, encore différente, pour "concrètement" répondre à la requête

Informatique - Mémento - Python avancé

Aide

`help(commande)` pour obtenir de l'aide sur une commande : `help(np.dot)` (`np.info` similaire).

Affectation

`x,y = exp1,exp2` : double affectation `x` reçoit `expr1` et `y` reçoit `expr2`,

On peut dépaqueter une liste avec une affectation multiple :

```
liste = [1,2,7]
a, b, c = liste
```

`x += a` : synonyme de `x = x + a`, existe aussi sous la forme `-=, *=, /=`,

Échange

Pour échanger 2 variables : `a, b = b, a`

Pour faire circuler 3 variables : `a, b, c = b, c, a`

Pour échanger deux éléments d'une liste :

```
L[i], L[j] = L[j], L[i]
```

Manipulation des entiers

`x//y` et `x%y` donnent le quotient et le reste de la division euclidienne de x par y. Le quotient est entier mais le reste est du même type que x : `3.54 % 2` donne 1.54 mais `3.54//2` donne 1

Numpy

`np.linspace (start, stop, num )` : tableau de `num` nombres régulièrement espacés sur `[start, stop]`

`np.arange(start, stop, step )` : tableau de nombres régulièrement espacés de `step` sur l'intervalle `[start, stop[`

Lecture d'un fichier

`f=open( nom du fichier,'r')` renvoie un objet fichier `f`

`f.readlines()` renvoie une liste de toutes les lignes d'un fichier (`"\n"` inclus)

`f.readline()` renvoie une ligne du fichier (`"\n"` inclus)

`for ligne in f:` parcourt toutes les lignes du fichier. Chaque ligne est une chaîne de caractères se terminant par `"\n"`

`f.close()` ferme le fichier

Tranches

`liste[a:b:s]` renvoie les éléments de la liste depuis `liste[a]` inclus à `liste[b]` exclu en sautant de `s` en `s`,

Par exemple, `liste[::-1]` renvoie la liste à l'envers.

Parcours de listes

`for k in range(len(liste)):` permet de parcourir les éléments d'une liste `liste` par leur indice `k`

`for e in liste:` permet de parcourir les éléments `e` d'une liste `liste`

Appartenance à une liste

`x in liste` permet de tester si x est un élément d'une liste `liste`

`x not in liste` pour le contraire ce qui permet d'écrire `if e not in L: L.append(e)`

Liste préremplie

`n * [0]` renvoie une liste de n zéros,

Listes en compréhension

`[ fonction(k) for k in range(n)]` renvoie la liste des valeurs `fonction(k)` pour k allant de 0 à $n-1$

Fonctions intrinsèques pour les listes

On peut utiliser toutes les fonctions intrinsèques de Python :

`len()`, `min()`, `max()`, `sum()`, `sorted()`, `reversed()`,

Les méthodes sur une liste `L` :

`L.append(x)`, `L.insert(i,x)`, `L.append(x)`, `L.pop(i)`, `L.sort()`, `L.reverse()`,

Et les fonctions de numpy : `np.min`, `np.max`, `np.sum`, `np.mean`, `np.std`, `np.conjugate`, ...

Chaines de caractères

`ch1 = ch1 + ch2` concaténation de chaînes ou `ch1 = ch2 + ch1` pour ajouter une chaîne à gauche,

`ch.strip()` supprime les espaces (et les fins de ligne) au début et à la fin de la chaîne,

`ch.split('x'))` découpe la chaîne en une liste de mots à chaque position de `x` (espace par défaut),

Booléens

Si `positif` est un booléen, on écrira `if positif:` ou `if not positif:` pour tester le booléen,

Pour le résultat d'une fonction, on écrira directement l'expression booléenne dans le return :

```
def pair(n):
    return n%2 ==0
```

Nombres complexes

`z = a +bj` est un nombre complexe et i est représenté par `1j`. On a les attributs `z.imag`, `z.real` et la méthode `z.conjugate()`

Mutabilité des listes

Les listes sont muables (ou mutables) et c'est pratique :

```
>>> L=[1,2,3]
>>> L[1]= 18
>>> L
[1, 18, 3]
```

Mais, ça a un revers ; la copie de listes n'est pas simple :

```
>>> L2=L
>>> L2, L
([1, 18, 3], [1, 18, 3])
>>> L2[1]=256
>>> L2, L
([1, 256, 3], [1, 256, 3])
```

Les fonctions peuvent modifier les listes par effet de bord :

```
>>> def f(L,M):
        L = [1,2,3]
        M[1] = 234

>>> L = [5,6,7]
>>> M = [5,6,7]
>>> f(L,M)
>>> L,M
([5, 6, 7], [5, 234, 7])
```

Copie de liste

en utilisant une tranche : `nvelle = ancienne[:]`

en utilisant une conversion :

`nvelle = list(ancienne)`

en utilisant le module `copy` :

```
import copy
nvelle = copy.copy(ancienne)
```

ou

```
import copy
nvelle = copy.deepcopy(ancienne)
```