Les algorithmes au programme

Attention ! Dans la suite on rappelle seulement les algorithmes au programme. Sont à réviser également : le fonctionnement basique d'un ordinateur, la représentation des réels et flottants en mémoire, les interactions avec un fichier, les bases de données.

1 Programme de première année

1.1 La base

Les algorithmes suivants sont de complexité $O(n)$ où n est la taille de la liste.

Recherche d'un élément présent dans une liste. Renvoie un booléen indiquant si `x` est présent dans `L`. S'obtient simplement par `x in L` en Python.

```
def cherche(L,x):  #le plus générique: on adapte le code facilement.
    n=len(L)
    for i in range(n):
        if L[i]==x:
            return True
    return False #attention erreur classique, pas de else : return False !

def cherche2(L,x): #parcours des éléments et pas des indices.
    for u in L:
        if u==x:
            return True
    return False
```

Maximum d'une liste non vide. S'adapte aussi pour le minimum, l'indice du maximum, etc... S'obtient simplement par `max(L)` en Python. `L.index(max(L))` en Python pour l'indice du maximum.

```
def maxi(L):
    n=len(L)
    m=L[0]
    for i in range(1,n):
        if L[i]>m
            m=L[i]
    return m
```

Somme des éléments d'une liste. Peut être utilisé pour calculer moyenne, variance, etc... `sum(L)` en Python.

```
def somme(L):
    n=len(L)
    s=0
    for i in range(n):
        s+=L[i]
    return s
```

Tester si une liste est croissante. À connaître !

```
def est_croissante(L):
    n=len(L)
    for i in range(n-1): #ne pas oublier le -1, sinon on dépasse !
        if L[i]>L[i+1]:
            return False
    return True
```

1.2 Recherche de motifs dans une chaîne de caractères

Il s'agit de chercher si les caractères de `m` apparaîssent contigument dans `s`.

```
def est_motif(m,s):
    n=len(s)
    k=len(m)
    for i in range(n-k+1):
        j=0
        while j<k and m[j]==s[i+j]:
            j+=1
        if j==k:
            return True
    return False
```

Remarque : le corps de boucle peut simplement être remplacé par le test `m==s[j:j+k]`. Complexité $O(kn)$. En Python, `m in s` fournit directement le résultat (plus rapidement !)

1.3 Recherche dichotomique/Résolution dichotomique

a) Recherche dichotomique dans une liste triée. On se donne une liste `L` croissante, et on cherche si un élément `x` est dans cette liste. Voici deux variantes proches (voir les invariants associés). Complexité $O(\log n)$ où n est la taille de la liste. Rappels : `L[:i]` désigne les éléments jusqu'à l'indice i exclus, `L[j:]` les éléments à partir de l'indice j inclus.

```
def cherche_dicho(L,x):
    i=0
    j=len(L)-1
    while i<=j:
    #Invariant: x n'est pas dans L[:i] ou L[j+1:]
        m=(i+j)//2
        if L[m]==x:
            return True
        elif L[m]<x:
            i=m+1
        else:
            j=m-1
    return False
```

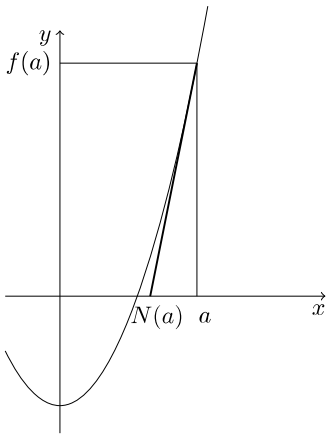
```
def cherche_dicho_2(L,x):
    i=0
    j=len(L)
    while i<j:
    #Invariant: x n'est pas dans L[:i] ou L[j:]
        m=(i+j)//2
        if L[m]==x:
            return True
        elif L[m]<x:
            i=m+1
        else:
            j=m
    return False
```

b) Résolution d'une équation numérique par dichotomie. Pour la recherche dichotomique, on se donne une fonction f continue sur $[a,b]$, avec $a<b$ et $f(a)f(b)\leq 0$. Le théorème des valeurs intermédiaires assure que f possède (au moins) un zéro dans l'intervalle, et on veut donner une approximation d'un tel zéro à ε près. À chaque étape, on coupe l'intervalle en deux, et on choisit l'intervalle qui maintient l'invariant $f(a)f(b)\leq 0$ (il en existe au moins un). Comme on renvoie $\frac{a+b}{2}$ à la fin de l'algorithme, on peut s'arrêter dès que $b-a\leq 2\varepsilon$.

```
def zero(f,a,b,eps):
    while b-a>2*eps:
        #Invariant: f(a)f(b)<=0
        m=(a+b)/2
        if f(m)*f(a)<=0:
            b=m
        else:
            a=m
    return (a+b)/2
```

1.4 Méthode de Newton

La méthode de Newton pour la résolution de $f(x)=0$ avec f une fonction $\mathcal{C}^1$ consiste à prendre un point de départ x_0 et itérer l'opération $a\mapsto\mathcal{N}(a)$ où $\mathcal{N}(a)$ est l'intersection de la tangente à f en a avec l'axe de abscisses :



Il faut savoir retrouver la formule : $\mathcal{N}(a) = a - f(a)/f'(a)$. La fonction suivante applique la méthode avec un nombre fixe d'itérations. Elle prend en paramètre la dérivée de f sous la forme d'une fonction g.

```
def Newton(f,g,x0,n):
    x=x0
    for i in range(n):
        x=x-f(x)/g(x)
    return(x)
```

Remarques : la méthode ne converge pas forcément, mais lorsque c'est le cas elle converge en général beaucoup plus vite que la méthode dichotomique : pour p étapes, on obtient une précision de l'ordre de 2^{-p} avec la méthode dichotomique contre 2^{-2^p} pour la méthode de Newton (sous de bonnes conditions). La méthode admet plusieurs variantes :

— on peut estimer la dérivée sous la forme $f'(x) \simeq \frac{f(x+h)-f(x)}{h}$ ou mieux $f'(x) \simeq \frac{f(x+h)-f(x-h)}{2h}$.

— la méthode s'étend en dimensions supérieures, la dérivée étend remplacée par la différentielle.

1.5 Pivot de Gauss

On rappelle juste l'algorithme ici. On suppose $A \in \mathcal{GL}_n(\mathbb{R})$ inversible (représentée par une liste de listes), on se donne un vecteur colonne Y (représenté comme une matrice de taille $n \times 1$, c'est-à-dire une liste de listes à un élément), et on cherche X tel que $AX = Y$. On commence par rendre A triangulaire supérieure par opérations sur les lignes, en faisant les mêmes opérations sur Y. Ensuite, résoudre un système triangulaire est facile. La complexité est $O(n^3)$ avec n la taille de la matrice.

Quelques fonctions auxiliaires

```
def echange_lignes(A,i,j): #on echange les deux lignes L_i et L_j.
    A[i], A[j] = A[j], A[i]

def transvection(A,i,j,mu): #L_i <- L_i+mu L_j
    n=len(A[0])
    for k in range(n):
        A[i][k]+=mu*A[j][k]

def indice_pivot(A,i):
    """ Renvoie l'indice du maximum en valeur absolue des éléments A[k][i], pour k>=i"""
    n=len(A)
    i_max=i
    for j in range(i+1,n):
        if abs(A[j][i])>abs(A[i_max][i]):
            i_max=j
    return i_max

def copie_matrice(A):
    return [x[:] for x in A]
```

Le pivot en lui-même

```
def gauss(A,Y): #renvoie le vecteur colonne X tel que AX=Y, on suppose qu'il existe.
    A, Y=copie_matrice(A), copie_matrice(Y)
    n=len(A)
    for i in range(n):
        imax=indice_pivot(A,i)
        if imax!=i:
            echange_lignes(A,i,imax)
            echange_lignes(Y,i,imax)
        for j in range(i+1,n):
            mu=-A[j][i]/A[i][i]
            transvection(A,j,i,mu)
            transvection(Y,j,i,mu)
    X=[[0] for i in range(n)]
    for i in range(n-1,-1,-1):
        X[i][0]=1/A[i][i]*(Y[i][0]-sum([A[i][j]*X[j][0] for j in range(i+1,n)]))
    return X
```

1.6 Intégration de fonctions numériques

On se donne $f : [a, b] \to \mathbb{R}$ une fonction continue, et on cherche à approcher $\int_a^b f(t)\,\mathrm{d}t$. La méthode consiste à donner d'abord un moyen d'approcher $\int_x^{x+h} f(t)\,\mathrm{d}t$ sur un petit intervalle $[x, x + h]$ puis appliquer cette méthode sur l'intervalle $[a, b]$ découpé en n morceaux.

a) Méthode des rectangles (à gauche) Elle consiste à faire l'approximation $\int_x^{x+h} f(t)\,\mathrm{d}t \simeq hf(x)$. On en déduit la fonction suivante, prenant en paramètre le nombre n de morceaux en lesquels on découpe $[a, b]$.

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

Cette méthode admet deux variantes : rectangle à droite et point milieu, avec les approximations $\int_x^{x+h} f(t)\,\mathrm{d}t \simeq hf(x + h)$ et $\int_x^{x+h} f(t)\,\mathrm{d}t \simeq hf\,(x + h/2)$. Pour f de classe $\mathcal{C}^2$, on peut montrer que l'erreur est de l'ordre de $O(1/n)$ pour la méthode des rectangles à gauche ou à droite (vrai si f est seulement $\mathcal{C}^1$) et de l'ordre de $O(1/n^2)$ pour la méthode du point milieu.

a) Méthode des trapèzes. Elle consiste à faire l'approximation $\int_x^{x+h} f(t)\,\mathrm{d}t \simeq h\frac{f(x)+f(x+h)}{2}$. On en déduit par exemple :

```
def int_trapz(f, a, b, n):
    s=0
    h=(b-a)/n
    x, y=a, a+h
    for i in range(n):
        s+=f(x)+f(y)
        x, y=y, y+h
    return s*h/2
```

L'erreur est d'ordre $O(1/n^2)$ si f est $\mathcal{C}^2$. Cette méthode est précise, et pratique si f est mesurée au lieu d'être calculée.

1.7 Méthode d'Euler

Il est impératif de savoir écrire une méthode d'Euler explicite (la plus simple). On cherche à résoudre le système

$$\begin{cases} x'(t) = f(x(t), t) \\ x(t_0) = x_0 \end{cases}$$

sur un intervalle $[t_0, t_{\text{lim}}]$. La méthode d'Euler explicite est basée sur la méthode d'intégration des rectangles à gauche : pour h petit, on a $x(t + h) = x(t) + \int_t^{t+h} x'(u)\,\mathrm{d}u \simeq x(t) + hx'(t) = x(t) + hf(x(t), t)$.

La fonction Python suivante prend en paramètre la fonction f, le pas h, deux réels t_0 et x_0 et le nombre total n de points à calculer, elle renvoie les temps $(t_i)_{0\leq i<n}$ et les valeurs approchées de x en ces points sous la forme de deux listes.

```
def euler(f, h, t0, x0, n):
    X=[x0]
    T=[t0]
    for i in range(n-1):
        x, t=X[i], T[i]
        X.append(x+h*f(x, t))
        T.append(t+h)
    return T,X
```

Cette fonction est très générique, mais il faut savoir adapter le principe au sujet. La méthode d'Euler n'est pas très précise, mais c'est la plus simple et la seule au programme.

2 Programme de deuxième année

2.1 Tris quadratiques

Les deux tris suivants prennent en paramètre une liste, et la trie en place. Ils sont de complexité $O(n^2)$ tous les deux, toutefois le tri par insertion a une complexité $O(n)$ sur des listes « presque triées ».

```
def tri_selection(L):
    n=len(L)
    for i in range(n-1):
        #Inv(i): L[0:i] triée, ses éléments sont plus petits que les autres éléments de L.
        imin=i
        for j in range(i+1,n):
            #Inv2(j): imin est l'indice du plus petit élément de L[i:j]
            if L[j]<L[imin]:
                imin=j
            #Inv2(j+1)
        if i!=imin:
            L[i],L[imin]=L[imin],L[i]
        #Inv(i+1)
```

```
def tri_insertion(L):
    n=len(L)
    for i in range(1,n):
        #Inv(i): L[0:i] est triée
        j=i
        x=L[i]
        while j>0 and L[j-1]>x:
            #Inv2: Pour tout k vérifiant j<k<=i, L[k] a été déplacé d'un cran à droite et est > x
            L[j]=L[j-1]
            j-=1
            #Inv2: Pour tout k vérifiant j<k<=i, L[k] a été déplacé d'un cran à droite et est > x
        L[j]=x
        #Inv(i+1): L[0:i+1] est triée
```

2.2 Tris efficaces

Ils sont tous deux récursifs. Le tri fusion renvoie une copie triée de la liste, alors que le tri rapide s'effectue en place (il modifie la liste mais ne renvoie rien). La complexité du tri fusion est $O(n \log n)$, celle du tri rapide est $O(n \log n)$ en moyenne seulement, car $O(n^2)$ dans le pire cas (correspondant par exemple à une liste triée).

a) Tri fusion. On utilise une fonction auxiliaire de fusion de deux listes triées.

```
def fusion(L1,L2):
    L=[]
    i1=0 ; i2=0
    n1=len(L1) ; n2=len(L2)
    for i in range(n1+n2):
        if i2==n2 or i1<n1 and L1[i1]<=L2[i2]:
            L.append(L1[i1])
            i1+=1
        else:
            L.append(L2[i2])
            i2+=1
    return L
```

```
def tri_fusion(L):
    if len(L)<=1:
        return L[:]
    else:
        m=len(L)//2
        return fusion(tri_fusion(L[:m]),tri_fusion(L[m:]))
```

b) Tri rapide. On utilise une fonction de partition d'une portion `L[g:d]` de la liste `L`. Celle-ci répartit les éléments autour du pivot `L[g]`, et renvoie son indice final dans la liste.

```
def partition(L,g,d):
    pivot=L[g]
    m=g
    for i in range(g+1,d):
        if L[i]<pivot:
            m+=1
            if i>m:
                L[i],L[m]=L[m],L[i]
    if m>g:
        L[m],L[g]=L[g],L[m]
    return m
```

Le tri rapide fait usage d'une fonction auxiliaire dont le but est de trier récursivement la portion `L[g:d]`.

```
def tri_rapide(L):
    def aux(g,d):
        if g<d-1: #sinon il n'y a rien a faire.
            m=partition(L,g,d)
            aux(g,m)
            aux(m+1,d)
    aux(0,len(L))
```