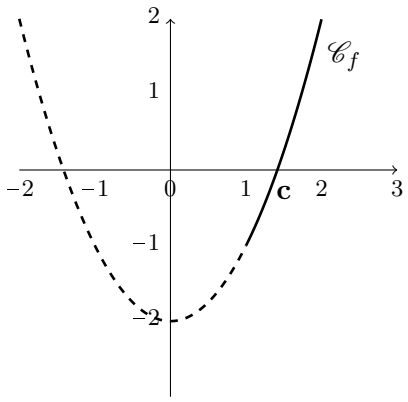
TP3 PCSI Python
Algorithmes dichotomiques

Dans ce chapitre, nous allons voir comment faire une recherche de manière beaucoup plus efficace qu'en essayant toutes les possibilités une à une.
Il existe en effet une stratégie générale, appelée *dichotomie* qui consiste à éliminer la moitié des possibilités à chaque étape.
Cette stratégie est utilisée aussi bien en mathématiques pour approcher une racine d'une fonction qu'en informatique pour rechercher rapidement une valeur dans une liste triée. Dans les deux cas, le principe est le même : diviser le problème par deux à chaque étape.

I)Valeur approchée de la racine d'une fonction

Considérons par exemple $f : x \mapsto x^2 - 2$ sur [1,2].



f est continue sur [1 ;2]. De plus, $f(1) = -1 < 0$ et $f(2) = 2 > 0$.

D'après le théorème des valeurs intermédiaires, il existe au moins un réel $c \in [1;2]$ tel que $f(c) = 0$.

De plus, f est strictement croissante sur [1 ;2], ce qui assure l'unicité de la racine c, d'après le théorème de bijection.

Invariant de boucle :
On construit une suite d'intervalles $[a;b]$ inclus les uns dans les autres avec $f(a)$ et $f(b)$ de signes contraires, ce qui garantit qu'à chaque itération la propriété « $c \in [a;b]$ » est préservée.

Principe de la dichotomie à chaque itération :
– on calcule le milieu m de $[a;b]$,
– on évalue f en m,
– on remplace $[a;b]$ par l'une des deux moitiés $[a;m]$ ou $[m;b]$, de sorte que f change de signe aux extrémités de l'intervalle.

Programme :
Ecrivons un programme qui affiche une valeur approchée de c à $0,2$ près.

```
def f(x):
    return x**2-2
a=1
b=2
epsilon=0.2
while b-a>epsilon:
    m=(a+b)/2
    if f(a)*f(m)<=0:
        b=m
    else:
        a=m
print(a,b)
```

Exécution :

boucle	m	$f(a)$	$f(m)$	intervalle contenant c	a	b	$b-a$
$\times\times\times$	$\times\times\times$	$\times\times\times$	$\times\times\times$	$\times\times\times\times\times\times\times\times\times\times$	1	2	1
1	3/2	-1	1/4	$c \in [a,m]$ et $m \longleftarrow b$	1	3/2	1/2
2	5/4	-1	$-7/16$	$c \in [m,b]$ et $a \longrightarrow m$	5/4	3/2	1/4
3	11/8	$-7/16$	$-7/64$	$c \in [m,b]$ et $a \longrightarrow m$	11/8	3/2	1/8

À chaque étape, la longueur de l'intervalle $[a,b]$ est divisée par deux, tout en continuant à contenir la racine c. On poursuit ainsi jusqu'à ce que la longueur de l'intervalle $[a,b]$ soit inférieure à la précision `epsilon` souhaitée.

Au bout de 3 itérations, $b-a = 1/8 < 0.2$.

On sort alors de la boucle et on obtient : $\frac{11}{8} \leq c \leq \frac{3}{2}$, soit $1.375 \leq c \leq 1.5$

Remarques
1)La condition `if f(a)*f(m)<= 0` regarde si $f(a)$ et $f(m)$ sont de signes contraires, auquel cas $c \in [a,m]$, sinon c'est que $c \in [m,b]$.
2)Lorsque b-a≤ epsilon, la racine c est alors dans un intervalle de longueur inférieure à la précision demandée, on peut donc arrêter l'algorithme.
3)Pour cet exemple, $c = \sqrt{2}$

II)Recherche dans une liste

On cherche à déterminer si une valeur x donnée est présente ou non dans une liste L. Nous allons étudier deux stratégies.

II.1)Recherche naïve

La recherche naïve est la plus naturelle, elle consiste à parcourir les éléments de la liste de gauche à droite.

- soit on y trouve la valeur x et on renvoie son indice,
- soit on atteint la fin de la liste sans jamais la trouver et on retourne -1.

```
def recherche(L,x):
    for i in range(len(L)):
        if L[i]==x:
            return i
    return -1
```

II.2)Recherche dichotomique dans une liste triée

Soit une liste L dont les éléments sont ordonnés dans **l'ordre croissant** (avec présence éventuelle de doublons).
On raisonne sur un intervalle [g,d] constitué d'indices consécutifs.

Invariant de boucle :
A chaque itération, la propriété suivante est préservée :
« Si x appartient à L, son indice appartient toujours à l'intervalle [g,d] ».

Principe de la dichotomie à chaque itération :
– on calcule l'indice médian de [g,d], noté m,
– on compare la valeur L[m] à la valeur recherchée x,
– si L[m]=x, la recherche est terminée. Sinon, on poursuit la recherche dans la seule moitié de la liste où x peut encore se trouver.
L'algorithme s'arrête dès lors que x est trouvé, on renvoie alors son indice. Si x n'a pas été trouvé, on renvoie -1.

Programme :

```
def recherche(L,x):
    g=0
    d=len(L)-1
    while g<=d:
        m=(g+d)//2
        if L[m]<x:
            g=m+1
        elif L[m]>x:
            d=m-1
        else:
            return m
    return -1
```

Remarque
Comme $\frac{g+d}{2}$ n'est pas toujours un entier, on définit l'indice médian par : $m = \left\lfloor \frac{g+d}{2} \right\rfloor$ ou ce qui revient au même par $m = (g+d)//2$.

Exécution :
Prenons L = [5,10,15,20,30,40,50,60,65,70,80]. On a : len(L)=11.
On cherche à savoir si la valeur $\boxed{x = 60}$ est présente dans cette liste.

boucle	m	$L[m]$	position de x	g	d
$\times\times\times$	$\times\times\times$	$\times\times\times$	$\times\times\times\times\times\times\times\times\times\times$	0	10
1	5	40	$L[m] < x$ donc $g \longrightarrow m+1$	6	10
2	8	65	$L[m] > x$ donc $m-1 \longleftarrow d$	6	7
3	6	50	$L[m] < x$ donc $g \longrightarrow m+1$	7	7
4	7	60	$L[m] = x$, on retourne 7	$\times\times\times$	$\times\times\times$

II.3)Comparaison des deux méthodes

La *complexité* d'un programme est le nombre d'opérations élémentaires qu'il effectue, ce nombre étant évalué en fonction de la taille n des données.
Pour notre étude, n est la longueur de la liste L.

Recherche naïve
Dans le pire des cas, il faut parcourir toute la liste L, ce qui nécessite n itérations de la boucle. À chaque itération, on effectue un nombre constant d'opérations élémentaires. La complexité du programme est donc en $O(n)$, c'est une *complexité linéaire*.

Recherche dichotomique
À chaque itération, le nombre d'indices restant à examiner est environ divisé par deux. On peut montrer que la complexité du programme est en $O\big(\ln(n)\big)$ du fait de la division par 2 à chaque itération (voir exercice 3), c'est une *complexité logarithmique*.

Conclusion
La recherche dichotomique dans une liste est beaucoup plus rapide que la recherche naïve. Elle a cependant deux désavantages :
– elle ne s'applique que si la liste est triée,
– elle ne fournit pas l'indice de la première occurence de x, contrairement à la recherche naïve, mais uniquement l'indice d'une occurence de x (on verra dans l'exercice 5 comment remédier à ce problème).

III)Exercices

Exercice 1★ ☆ ☆☆

1)En utilisant la méthode de dichotomie avec une fonction f bien choisie et définie sur [1,2], compléter le programme suivant afin qu'il affiche une valeur approchée de $c = \sqrt[3]{5}$ à 10^{-6} près.

```
def f(x):
    return .....
a=1
b=2
epsilon=10**-6
while b-a>epsilon:
    m=(a+b)/2
    if f(a)*f(m)<=0:
        .....
    else:
        .....
print(a,b)
```

2)Compléter le tableau suivant :

boucle	m	$f(a)$	$f(m)$	intervalle contenant c	a	b	$b-a$
××	××	××	××	××	1	2	1
1							
2							

3)Soit k le nombre total d'itérations de la boucle `while`.

Justifier que $\left(\frac{1}{2}\right)^k \leq 10^{-6} < \left(\frac{1}{2}\right)^{k-1}$, puis déterminer k.

4)Retrouver ce résultat en introduisant un compteur dans la boucle.

Exercice 2★ ☆ ☆☆

Soit f une fonction continue et strictement monotone sur [a,b] vérifiant $f(a)f(b) < 0$. On note c l'unique racine de f.

Pour trouver une valeur approchée de c, on procède par balayage et non plus par dichotomie.

On pose $h = \frac{b-a}{n}$ où $n \in \mathbf{N}^*$.

On examine successivement les valeurs de $f(a)$, $f(a+h)$, $f(a+2h)$, ..., $f(b)$ jusqu'à ce qu'on observe un changement de signe.

1)Compléter les deux instructions manquantes de la fonction suivante afin qu'elle renvoie les extrémités d'un intervalle de longueur h où se trouve c.

```
def balayage(a,b,n):
    h=(b-a)/n
    x=a
    while f(x)*f(x+h)>0:
        ...........
    return .........
```

2)Utiliser cette fonction pour trouver une valeur approchée de $\sqrt[3]{5}$ à 10^{-6} près. Combien d'itérations sont nécessaires ? Comparer avec l'exercice 1.

Exercice 3★ ★ ☆☆

Dans la fonction suivante d'argument $n \in \mathbf{N}^*$, on cherche à évaluer le nombre total p d'itérations de la boucle `while`.

```
def dicho(n):
    while n>0:
        n=n//2
```

1)Pour tout $i \in [\![0,p]\!]$, on note u_i la valeur de la variable `n`, à l'issue de i itérations de la boucle.
On a donc $u_0 = n$ et $\forall i \in [\![0,p-1]\!],\ u_{i+1} = u_i//2$.

a)Que valent u_p et u_{p-1} ?

b)Justifier que $\forall i \in [\![0,p-1]\!],\ 0 \leq 2^i u_i - 2^{i+1}u_{i+1} \leq 2^i \quad (*)$

c)En sommant les inégalités $(*)$ pour i allant de 0 à $p-2$, établir :

$$2^{p-1} \leq n \leq 2^p - 1.$$

d)En déduire que $p = \lfloor \log_2(n) \rfloor + 1$.

2)Modifier la fonction dicho afin qu'elle retourne le nombre total d'itérations de la boucle while. Appeler cette nouvelle fonction pour $n = 10000$.
Le résultat obtenu est-il conforme à celui de la question 1)d) ?

Exercice 4★ ★ ☆☆

1)Programme :

```
def recherche(L,x):
    g=....
    d=....
    while g<=d:
        m=....
        if L[m]<x:
            .........
        elif L[m]>x:
            .........
        else:
            return ..... #j'ai trouvé x
    return ...... #je n'ai pas trouvé x
```

La fonction `recherche` prend comme arguments une liste L **croissante** d'entiers et un nombre x.
En utilisant le cours, compléter cette fonction de sorte qu'elle renvoie l'indice d'une occurence de x dans la liste L , si x s'y trouve, ou qui renvoie -1 sinon.
La recherche doit se faire par dichotomie.

2)Pour chacun des cas suivants, compléter le tableau, puis préciser la valeur renvoyée par la fonction.

a)L=[1,3,7,12,15,20,30] et x=15.

boucle	m	$L[m]$	position de x	g	d
$\times\times\times$	$\times\times\times$	$\times\times\times$	$\times\times\times\times\times\times\times\times\times\times$	0	6
1					
2					
3				$\times\times\times$	$\times\times\times$

b)L=[2,3,5,5,8,10,15,20,30,40] et x=5

boucle	m	$L[m]$	position de x	g	d
$\times\times\times$	$\times\times\times$	$\times\times\times$	$\times\times\times\times\times\times\times\times\times\times$	0	9
1					
2					
3				$\times\times\times$	$\times\times\times$

c)L=[1,4,7,11,15,20,30,50] et x=10

boucle	m	$L[m]$	position de x	g	d
$\times\times\times$	$\times\times\times$	$\times\times\times$	$\times\times\times\times\times\times\times\times\times\times$	0	7
1					
2					
3					

3)Appeler la fonction avec L=[5,7,7,7,7,7] et x=7.
Retourne t-elle l'indice de la première occurence de 7 ?

Exercice 5★ ★ ★☆

On cherche à améliorer la fonction `recherche` de l'exercice 4 afin qu'elle renvoie maintenant l'indice de la **première occurence** de x dans la liste croissante L, si x s'y trouve, et qui renvoie -1 sinon.

1)On propose la fonction suivante :

```
def premoccurence1(L,x):
    g=0
    d=len(L)-1
    while g<=d:
        m=(g+d)//2
        if L[m]<x:
            g=m+1
        elif L[m]>x:
            d=m-1
        else:
            while L[m]==x:
                m=m-1
            return m+1
    return -1
```

a)Appeler cette fonction avec les cas vus en 2) de l'exercice 4. Vérifier qu'elle retourne bien la bonne réponse.

b)On appelle cette fonction avec L=[1,1,1,1,1,1] et x=1. Python renvoie le message d'erreur : `list index out of range`. D'où vient cette erreur ?

c)A la place de `while L[m]==x`, on met : `while m>=0 and L[m]==x`
Le message d'erreur n'apparaît plus.
Cette nouvelle fonction répond au problème, mais elle a un inconvénient au niveau de sa complexité. Pourquoi ?

2)On considère la fonction suivante :

```
def premoccurence2(L,x): # CCINP TSI 2020
    g=0
    d=len(L)-1
    while g<d:
        m=(g+d)//2
        if L[m]<x:
            g=m+1
        else:
            d=m
    if L[g]==x:
           return g
    else:
        return -1
```

a)Si on remplace la condition `g<d` de la boucle `while` par `g<=d`, comme dans la recherche dichotomique classique, que risque t-il de se passer ?

b)On suppose que x∈ L. On note i l'indice de la première occurence de x.

b1)Vérifier qu'avant la première itération de la boucle, on a bien $i \in [g, d]$.

b2)Montrer que si $i \in [g, d]$ en début de boucle, on a encore $i \in [g, d]$ en fin de boucle. La propriété « $i \in [g, d]$ » est donc un invariant de boucle.

c)Comparer les valeurs de g et d à la sortie définitive de la boucle `while`.

d)Conclure que si x appartient à L, le programme renvoie nécessairement l'indice de la première occurrence de x, et non pas simplement l'indice d'une occurrence quelconque comme dans l'exercice 4.

e)Après le test `else`, pourquoi n'écrit-on pas `d=m-1` à la place de `d=m` ?

f)On donne : L=[0,0,0,1,1,1,1,1,1,2,2,2,2,6,9] et x=2. Compléter le tableau :

boucle	m	$L[m]$	position de x	g	d
×××	×××	×××	××××××××××	0	13
1					
2					
3					
4				8	8

Exercice 6★ ★ ☆☆

```
def devine():
    debut=1
    fin=100
    while debut<=fin:
        m=(debut+fin)//2
        print("Est-ce", m, "?")
        reponse=input("trop grand (G), trop petit (P) ou trouvé (T) ? ")
        if reponse=="T":
            return m
        elif reponse=="P":
            debut=m+1
        else:
            fin=m-1
```

1)But du programme ? Principe utilisé pour trouver le nombre choisi ?

2)On choisit le nombre 60.

a)Exécuter le programme. Combien comporte t-il d'itérations ?

b)Est-il aussi efficace de choisir m au hasard entre `debut` et `fin` ?

Indication : on pourra utiliser la commande `randint(a,b+1)` du module `numpy.random` qui renvoie un entier aléatoire entre a et b.

Exercice 7★ ★ ★☆

Soient deux listes L et M triées dans l'ordre **strictement croissant**, et de longueurs respectives n et p. On veut écrire une fonction prenant comme arguments L et M et renvoyant la liste des éléments communs à L et M.

1)Compléter la fonction ci-dessous de sorte qu'elle réponde au problème :

```
def communs1(L,M):
    N=[]
    for x in M:
        for y in L:
            if x==y:
                .....
    return .....
```

b)Justifier que la complexité de la fonction `communs1` est en $O(np)$.

2)a)Ecrire la fonction demandée d'en-tête `def communs2(L,M):`
en considérant maintenant l'algorithme suivant :
– on parcourt les p éléments de M,
– pour tout élément x de M, on fait une recherche dichotomique dans L, en utilisant la fonction `recherche` de l'exercice 4,
– si besoin, on ajoute x à la liste des éléments communs à L et M.

b)Quelle est la complexité de la fonction `communs2` ?

3)a)On souhaite maintenant écrire la fonction demandée, mais sans utiliser de dichotomie, en comparant L[i] et M[j]. Compléter :

```
def communs3(L,M):
    N=[]
    i=0
    j=0
    while i<len(L) and j<len(M):
        if L[i]<M[j]:
            i=.....
        elif L[i]>M[j]:
            j=.....
        else:
            N.append(.....)
            i=.....
            j=.....
    return N
```

b)Quelle est la complexité de la fonction `communs3` ?

4)Comparer les complexités des trois fonctions `communs1`, `communs2` et `communs3`. Laquelle des trois méthodes est la plus avantageuse en terme de coût ?