CORRECTION DU TD IPT² N° 2: RÉVISIONS 2/2
PREUVE ET COMPLEXITÉ DES ALGORITHMES - EXTRAITS DE CONCOURS

Preuve et complexité d'algorithmes

EXERCICE N°1: Complexité et preuve d'un algorithme de calcul de série

❶ Le script demandé est en fait le programme proposé en *listing 4* un peu plus bas dans l'énoncé.

❷ **Analyse de scripts**

a· **Algorithme de gauche**:
Appelons C la condition $C(x, s) = (s \leq x)$.

- à l'entrée dans la boucle s contient 1 et n contient 1 donc la condition C est vérifiée.
- Supposons qu'au rang k, n contienne k et s contienne s_k, alors:
  - soit $C_k(x, s_k)$ n'est pas vérifiée et le programme termine en affichant $n = k = N$ et s_k.
  - soit n est incrémenté et contient alors $k + 1$ et on affecte à s la valeur $s_k + \dfrac{1}{k+1} = s_{k+1}$; c'est bien la valeur attendu de la série au rang suivant.

  En outre, la série calculée diverge ($\lim_{k\to+\infty} s_k = +\infty$) ainsi, il existe un rang k_{max} pour lequel la condition $C(x, k_{max})$ sera violée alors qu'elle ne l'était pas au rang précédent $k_{max} - 1$. Donc arrivé à k_{max}, le programme termine en affichant bien la valeur de la série au rang $k_{max} = N$ attendu qui assure $s_N > x$.

Algorithme de droite: **NB:** dans cette version, la série est recalculée jusqu'au rang n à chaque incrémentation de n. L'analyse de preuve est cependant la même que précédemment.

b· En fait, il arrive un certain rang N_l dans le calcul de cette série pour lequel le terme ajouté est évalué à 0 par la machine (dépend de la profondeur de codage des flottants cf chapitre 3), ainsi la série ne diverge plus et si $s_{N_l} \leq x$ alors le programme ne termine pas.

c·

❸ ALGORITHME DE GAUCHE

- pour atteindre N le programme effectue $N - 1$ additions pour l'incrémentation de n et $N - 1$ additions pour l'ajout du terme de la série, soit un total de $2(N - 1)$ additions.
- Le nombre de divisions est alors $N - 1$.

ALGORITHME DE DROITE

Cette fois la boucle `while` est toujours réalisée $N - 1$ fois et réalise $N - 1$ additions, mais à la $n^{\text{ième}}$ itération, la boucle `for` exécute $n - 1$ additions et $n - 1$ divisions.

On a donc:

$$\sum_{n=1}^{N-1}(n-1) = (N-1)\times\frac{N-2}{2} \text{ divisions}$$

$$(N-1)\times\frac{N-2}{2} + (N-1) = \frac{N}{2}(N-1) \text{ additions}$$

Conclusion: le programme de droite est donc moins efficace en raison de calculs répétés inutilement.

❹ Le programme qui suit procède de la même façon que le programme de droite en évaluant systématiquement le valeur de la série jusqu'au rang k à la $k^{\text{ième}}$ itération. Il n'est donc pas très performant; en revanche, l'utilisation d'un tableau `numpy` et de méthodes intégrées comme `sum` doit sensiblement améliorer le temps de calcul.

EXERCICE N°2: Tri naïf

Listing 1:

1.
```python
def max(L,deb):
    rang,maxi=deb,L[deb]
    for pos in range(deb+1,len(L)):
        if L[pos]>maxi:
            rang,maxi=pos,L[pos]
    return rang,maxi
```

2. **a·** Dans ce script, l'argument L désigne une liste donc un objet mutable modifié au cours de l'exécution du programme. La liste L n'est pas renvoyée en fin de fonction, cependant, son contenu a été modifié par la fonction et sa nouvelle version est accessible depuis l'extérieur de la fonction puisque son étiquette L n'est pas un contenant mais pointe vers son emplacement mémoire. La commande `None` permet de terminer la fonction sans renvoi de variable.

   b· Le script exploite une boucle inconditionnelle `for` dont le variant k prendra en fin d'itération la valeur $len(L)-2$ (valeur finie); la fonction termine donc.

   c· Montrons que la propriété est vraie par exemple au rang 1 après une itération de boucle:

$$\begin{cases} L_1[0] \leq L_1[0] \text{ vrai après 1ère itération car égalité} \\ L_1[i] \leq L_1[0] \forall i \geq 1 \text{ vrai après 1ère itération} \\ \text{car max détecte le maximum de la liste entière qui se place ensuite en position d'indice } k=0 \end{cases}$$

Supposons la propriété vraie au rang j et montrons qu'elle est héréditaire ie. valable au rang $j+1$.
Après j itérations les j premiers nombres sont classés par ordre décroissant.
La $j+1^{\text{ième}}$ itération place le maximum de $L[j]\dots L[n-1]$ en $L_{j+1}[j]$ donc:

$$\begin{cases} L_{j+1}[0] \geqslant L_{j+1}[1] \geqslant \dots L_{j+1}[j-1] \geqslant L_{j+1}[j] \\ \forall i \geqslant j+1 \quad L_{j+1}[i] \leqslant L_{j+1}[j] \text{ puisque le maximum de } L[j]\dots L[n-1] \text{ est maintenant en } L_{j+1}[j] \end{cases}$$

Ceci est bien la propriété au rang $j+1$ suivant. $\mathcal{P}_j$ est donc un invariant de boucle.

   d· Nombre d'itérations de l'algorithme:

- Pour la $1^{\text{ère}}$ itération $k=0$ de la première boucle, la boucle de max exécute $n-1$ itérations
- Pour la 2^{nde} itération $k=1$ de la première boucle, la boucle de max exécute $n-2$ itérations
- ...
- **Pour la $k^{\text{ième}}$ itération de la première boucle, la boucle de max exécute $(n-k-1)$ itérations**
- ...
- Pour la $n-1^{\text{ième}}$ itération $k=n-2$ de la première boucle, et dans la boucle de la fonction `max` l'indice `pos` varie de $n-1$ à $n-1$ inclus, donc réalise une seule itération

Finalement, le nombre total d'itérations est:

$$\sum_{k=0}^{n-2} n-k-1 = (n-1)+(n-2)+\dots\underbrace{(n-(n-2)+1)}_{=1}$$

$$= (1+2+3+\dots+n-1) = (n-1)\frac{n}{2}$$

ce qui permet de conclure à une complexité asymptotique $C(n) = O(n^2)$

Exercice n°3: Recherche dichotomique

1. Plusieurs algorithmes sont possibles suivant que l'on souhaite exploiter certaines fonctions pratiques de Python ou pas:

Première proposition: à l'aide de l'instruction `in`

Listing 2:

```python
def app(e,T):
        if e in T:
        return True
    else:
        return False
```

Seconde proposition: avec boucle inconditionnelle

Listing 3:

```python
    def app(e,T):
        for pos in range(len(T)): # attention: boucle for qui ne termine pas dans la cas général-> mauvaise pratique mais acceptable à l'écrit.
            if T[pos]==e:
                return True
        return False

```

Troisième proposition: avec boucle conditionnelle

Listing 4:

```python
    def app(e,T):
    pos=0
    while (pos!=len(T)):
        if T[pos]==e:
            return True
        else:
            pos=pos+1
    return False

```

2. Le nombre maximum d'itérations est obtenu lorsque l'élément recherché se trouve en dernière position de la liste, ainsi les boucles conditionnelle et inconditionnelle auront effectué `Nb=len(T)` itérations.

3. **a**· On propose les modifications d'algorithme suivantes:

Listing 5:

```python
def dicho(e,T):
    g, d = 0, len(T)-1
    while g <= d:
        m = (g + d) // 2
        if T[m] == e:
            return True
        if T[m] < e: #e est dans le tableau de droite
            g=m+1
        else: #e est dans le tableau de gauche
            d=m-1
    return False
```

b· On montre facilement la proposition par récurrence:

Supposons la propriété vraie à un rang i, par exemple au rang 1 après une itération (avec coupure à gauche ou à droite) puisque:
$d_1 - g_1 \overset{d_1=m-1}{=} \left(\left\lfloor \frac{g_0+d_0}{2}\right\rfloor -1\right) - g_0 < \frac{d_0+g_0}{2} - g_0 = \frac{d_0-g_0}{2} < \frac{n}{2}$
et avec coupure à gauche:
$d_1 - g_1 \overset{g_1=m+1}{=} d_0 - \left(E\left[\frac{g_0+d_0}{2}\right]+1\right) < d_0 - \frac{g_0+d_0}{2} = \frac{d_0-g_0}{2} < \frac{n}{2}$
Montrons l'hérédité: au rang $i+1$, on a:

$$d_{i+1}-g_{i+1} \overset{d_{i+1}=m-1}{=} \left(\left\lfloor\frac{g_i+d_i}{2}\right\rfloor -1\right) - g_i < \frac{d_i+g_i}{2} - g_i = \frac{d_i-g_i}{2} < \frac{1}{2}\frac{n}{2^i} < \frac{n}{2^{i+1}}$$

et idem avec coupure à gauche.
ceci est bien la propriété au rang $i+1$, prouvant la proposition.

c· Nombre maximum d'itérations:
Désignons par N_{max} le nombre maximum d'itérations, i.e. la solution sera trouvée lorsque la longueur de la sous-liste sera réduite à 0; si l'algorithme n'a pas encore renvoyé de solutions à $N_{max}-1$ itérations alors on a:

$$d_{N_{max}-1} - g_{N_{max}-1} = 1 < \frac{n}{2^{N_{max}-1}}$$

qui donne finalement:

$$\boxed{N_{max} < \frac{\ln n}{\ln 2} + 1 = \ln_2 n + 1}$$

d· Terminaison de l'algorithme:

- Si la solution est dégagée à un rang inférieur au rang maximum d'itérations avec $T[m]=e$, l'algorithme renvoie `True` et termine.
- Si la solution n'est pas dégagée au rang N_{max}, rang maximum d'itérations qui correspond à la situation $d-g=0$, alors on execute une itération de plus et:
  - soit l'élément est dans la liste, avec $T[m]=e$ le programme renvoie `True`
  - soit l'élément n'est pas dans la liste est l'on décrémente d'une unité l'écart $d-g$ avec les commande $m-1$ ou $m+1$ suivant la position de l'élément recherché e par rapport à $T[m]$. La condition de boucle est alors violée avec $d-g=-1$ et l'algorithme termine.

e· Au rang 0 avant itération la propriété est vérifiée puisque $g_0 < d_0$ et l'élément étant dans la liste $T[g_0] \leq e \leq T[d_0]$.
Supposons la propriété vérifiée lorsque l'on rentre dans la $k^{\text{ième}}$ itération avec $\begin{cases} g_k \leq d_k \\ T[g_k] \leq e \leq T[d_k] \end{cases}$
Montrons que la propriété est héréditaire en entrant dans la $k+1^{\text{ième}}$ itération, c'est à dire si $T[m_k] \neq e$. On a forcément $g_{k+1} \leq d_{k+1}$ puisque l'on est entré dans la boucle.

- Soit à ce rang $T[m_{k+1}] < e$ et on a $T[m_{k+1}+1 = g_{k+1}] \leq e$ (on doit maintenant envisager l'égalité côté gauche) et comme $d_{k+1} = d_k$ soit $e \leq T[d_{k+1}]$ on a finalement:
$$T[g_{k+1}] \leq e \leq T[d_{k+1}]$$
- Soit à ce rang $T[m_{k+1}] > e$ et on a $T[m_{k+1}-1 = d_{k+1}] \geq e$ (on doit maintenant envisager l'égalité côté droit) et comme $g_{k+1} = g_k$ soit $T[g_{k+1}] \leq e$ on a finalement:
$$T[g_{k+1}] \leq e \leq T[d_{k+1}]$$

Dès que l'égalité est vérifiée à gauche ou a droite, la solution est trouvée $T[m]=e$ et l'algorithme termine avant la violation de condition de boucle en renvoyant `True`: il est prouvé.
NB: dans l'hypothèse $e \notin T$ (non envisagée dans l'énoncé) la condition de boucle finit par être violée et l'algorithme termine en renvoyant `False`: il est prouvé

Extraits de concours

Exercice n°4: Modèle microscopique d'un matériau magnétique (d'après CCMP)

❶ On importe les fonctions `exp` et `tanh` du module `math` avec:

Listing 6:

```python
from math import exp,tanh
from random import randrange,random
```

❷ La fonction `initialisation()` doit simplement construire une liste de $n = h^2$ *spins* up:

Listing 7:

```python
def initialisation():
    return [1]*n
```

ou encore (si l'on veut faire usage d'une boucle)

Listing 8:

```python
def initialisation():
    s=[]
    for i in range(n):
        s.append(1)
    return s
```

❸ On propose pour la fonction `initialisation_anti()`

Listing 9:

```python
def anti_initialisation():
    sgn=1
    s=[]
    for i in range(h):
        for j in range(h):
            s.append(sgn)
        sgn=-sgn
    return s
```

ou encore la version plus synthétique suivante,puisque h est pair:

Listing 10:

```python
def anti_initialisation():
    return ([1]*h+[-1]*h)*(h//2)
```

On peut enfin faire appel à la méthode `.extend` dont la particularité est d'itérer sur une liste passée en argument afin d'ajouter les éléments un à un à la liste à construire:

Listing 11:

```python
def initialisation_anti() :
    Lres=[]
    for i in range(h) :
        Lres.extend([(-1)**i]*h)
    return Lres
```

❹ On peut procéder par slicing avec:

Listing 12:

```python
def repliement(s):
    s1=[]
    for i in range(0,n,h):
        s1.append(s[i:i+h])
    return s1
```

ou encore avec 2 boucles:

Listing 13:

```python
def repliement(s):
    s1=[]
    for i in range(h):
        si=[]
        for j in range(h):
            si.append(s[i*h+j])
        s1.append(si)
    return s1
```

❺ On propose pour la fonction `liste_voisins(i)`:

Listing 14:

```python
def liste_voisins(i):
    L=[]
    ind_ligne=i//h #indice de la ligne de l'élément d'indice i dans la liste
    ind_colonne=i%h #indice de la colonne de l'élément d'indice i dans la liste
    if ind_colonne==0: #si élément sur la première colonne
        L.append(i+h-1) #on prend pour voisin de gauche l'élément en fin de ligne
    else:
        L.append(i-1)  #sinon le voisin de gauche immédiat
```

```
    if ind_colonne==h-1: #si l'élément est sur la dernière colonne
        L.append(i-(h-1)) #on prend pour voisin de droite l'élément en début de ligne
    else:
        L.append(i+1) #sinon le voisin de droite immédiat
    if ind_ligne==h-1: #si l'élément est sur la dernière ligne
        L.append(i%h) #on prend comme voisin de dessous l'élément en début de colonne
    else:
        L.append(i+h) #sinon le voisin de dessous immédiat
    if ind_ligne==0: #si  l'élément est sur la première ligne
        L.append(h*(h-1)+i) #on prend comme voisin de dessus l'élément en fin de colonne
    else:
        L.append(i-h) #sinon le voisin de dessus immédiat
    return L
```

❻ On propose pour la fonction `energie(s)`:

Listing 15:

```
def energie(s):
    E=0
    for i in range(n):
        liste_voisins_de_i=liste_voisin(i)
        for j in liste_voisins_de_i:
            E+=s[i]*s[j]
    return -E/2
```

❼ Pour la fonction `test_boltzmann(delta_e,T)` on peut par exemple proposer le code très explicite suivant:

Listing 16:

```
def test_boltzmann(delta_e,T):
    if delta_e<=0:
        return True
    else:
        p=exp(-delta_e/T)
        if random()<=p:
            return True
        else:
            return False
```

ou un code plus efficace évitant un branchement par rapport à la proposition ci-dessus:

Listing 17:

```
def test_boltzmann(delta_e,T):
    p=exp(-delta_e/T)
    if delta_e<=0 or random()<=p:
        return True
    return False
```

❽ Il s'agit ici d'évaluer la complexité asymptotique des deux algorithmes; pour la proposition `calcul_delta_e1`:

- la recopie de la liste s (en s_2) se fait en complexité linéaire: $O(n)$, puis
- l'inversion de signe du spin s_i est en complexité constante: $O(1)$, puis
- l'appel à deux reprises au code `energie`, avec 4n itérations pour chaque, est se fait donc en complexité linéaire: $O(n)$
- Enfin le renvoi de `delta_e` se fait en complexité constante: $O(1)$

La complexité est donc $C(n) = O(n)$.

Pour la seconde proposition `calcul_delta_e2`: en remarquant que l'écart d'énergie entre les deux configurations avant et après inversion du spin s_i provient simplement de la modification de l'énergie d'interaction entre s_i et chacun de ses voisins; si l'on reprend la double sommation, il y a 8 termes $s_i \times s_j$ modifiés on constate que le second code calcule bien la même chose avec une complexité très nettement améliorée: il y a seulement 4 voisins, ce qui aboutit à une complexité constante $O(1)$ pour ce second code qui est donc nettement plus efficace.

Quelques explications sur le code: avec 4 voisins avec le *spin* s_i, les termes de couplage **modifiés** $s[i] \times s[j]$ vont chacun être évalués 2 fois lorsque l'on déroule la sommation ce qui explique la présence du terme 2 dans la sommation. (en revanche, l'auteur semble avoir oublié le terme d'intégrale de couplage avec J=1)

❾ Cette question est assez élémentaire; on donne par exemple:

Listing 18:

```
def monte_carlo(s,T,n_tests):
    for k in range(n_tests):
        i=randrange(n)
        if test_boltzmann(calcul_delta_e2(s,i),T):
```

```python
                        s[i]=-s[i]
        return None
```

❿ Cette dernière question exploite l'ensemble des codes rédigés plus haut; on propose:

Listing 19:

```python
def aimantation_moyenne(n_tests,T):
        s=initialisation()
        monte_carlo(s,T,n_tests)
        somme_spin=0
        for si in s:
                somme_spin+=si
        return somme_spin/n
```

Exercice n°5: Modèle de déplacement des dunes par automates cellulaires (extrait CCMP)

❶ La fonction `random()` renvoyant un flottant dans l'intervalle [0.0, 1.0[on a donc:

$$1 \leq n < \frac{h}{2} + 2$$

❷ Avec la relation donnant arbitrairement n, le calcul est immédiat:

Listing 20:

```python
import random as rd
def calcul_n(h):
        if h>1:
                return int((h+2.00)/2*rd.random())+1
        return 0
```

❸ La variable `piles` est de type `list` et contient $P+1$ piles chacune représentée par sa hauteur h. Initialement, les piles sont toutes de hauteur nulle $h = 0$:

Listing 21:

```python
def initialisation(P):
        return [0]*(P+1)
```

❹ On propose la fonction suivante qui examine les piles une à une, en retirant à chaque fois de la pile en cours de traitement le nombre de grains tombé(s) sur la pile immédiatement à droite après l'avoir calculé, et en les ajoutant à cette dernière:

Listing 22:

```python
def actualise(piles,perdus):
        N=len(piles) # nombre de piles
        for i in range(1,N):
                n=calcul_n(piles[i-1]-piles[i])
                piles[i-1]-=n
                if i==N-1:
                        perdus+=n
                else:
                        piles[i]+=n
        return (piles,perdus)
```

❺ Pour le programme principal, on peut proposer:

Listing 23:

```python
perdus=0
P=int(input(u"nombre de piles du tas?"))
piles=initialisation(P)
piles[0]=1
while perdus<1000:
        exec=0
        while exec<10:
                perdus=actualise(piles,perdus)[1]
                exec+=1
        piles[0]+=1
```

❻ On peut par exemple tracer la hauteur de chaque pile à l'aide de la fonction `scatter` du module `matplotlib`; pour cela on examine chaque pile pour en déduire le nuage de points à tracer:

Listing 24:

```python
from matplotlib import pyplot as plt
.....# Codes précédents
N=len(piles)
X,Y=[],[]
for x in range(N-1): #inutile d'étudier le contenu de la dernière pile P+1 qui reste tjrs vide
        for y in range(1,piles[x]+1): # on itère sur tous les entiers entre 1 et la hauteur totale de la pile traitée
                X.append[x] # on ajoute à la liste des abscisses celle du point en cours de traitement dans pile[x]
                Y.append[y] # on ajoute à la liste des ordonnées celle de ce point
```

```python
plt.grid(color='blue',linestyle='-',linewidth=0.06) #pour
    insérer une grille
plt.axis('equal') # et pour qu'elle soit orthonormée!
plt.scatter(X,Y) # on trace le nuage de points
plt.show()
```

Exercice n°6: Modélisation d'une propagation virale par automate cellulaire (extrait CCMP)

❶ La fonction `grille(n)` renvoie une liste comportant n listes chacune contenant n fois 0.

❷ On propose:

Listing 25:

```python
def init(n):
        G=grille(n)
        Linfect=random.randrange(n)
        Cinfect=random.randrange(n)
        G[Linfect][Cinfect]=1
        return G
```

❸ Fonction élémentaire ici qui doit simplement balayer la grille pour recenser les 4 états possibles, soit:

Listing 26:

```python
def compte(G):
        n=len(G)
        L=[0,0,0,0]
        for l in range(n):
                for c in range(n):
                        if G[l][c]==0: L[0]+=1
                        if G[l][c]==1: L[1]+=1
                        if G[l][c]==2: L[2]+=1
                        if G[l][c]==3: L[3]+=1
        return L

```

Un autre méthode plus fine et de complexité un peu réduite consiste à exploiter un recensement des fréquences d'apparition de chaque état:

Listing 27:

```python
def compte(G):
        n=len(G)
        L=[0,0,0,0]
        for l in range(n):
                for c in range(n):
                        L[G[l][c]]+=1
        return L

```

❹ La fonction `est_exposee(G,i,j)` renvoie un `booléen`.

❺ pour les lignes 12 et 20 cela donne:

Listing 28:

```python
1                def est_exposee(G, i, j):
2                .......
3                .......
4                .......
5                .......
6                .......
7                .......
8                .......
9                .......
10               .......
11               .......
12               return (G[0][j-1]-1)*(G[1][j-1]-1)*(G[1][j]-1)
     *(G[1][j+1]-1)*(G[0][j+1]-1)==0
13               .......
14               .......
15               .......
16               .......
17               .......
18               .......
19               .......
20               return (G[i+1][j]-1)*(G[i+1][j+1]-1)*(G[i][j
     +1]-1)*(G[i-1][j+1]-1)*(G[i-1][j]-1)*(G[i-1][j-1]-1)*(G[i][
     j-1]-1)*(G[i+1][j-1]-1) ==0
21
```

❻ On propose pour cette fonction de balayer chacune des cases (i, j) de la grille si la case est saine, après vérification de son état d'exposition par la fonction `est_exposee(G,i,j)` on peut la faire basculer dans l'état infecté à l'aide de la fonction `bernouilli(p2)`; pour une case à l'état infecté, toujours à l'aide de la fonction `bernouilli(p1)`, on peut la faire basculer à l'état décédé ou rétabli; cela donne:

Listing 29:

```
def suivant(G,p1,p2):
        n=len(G)
        Gprime=grille(n)
        for i in range(n):
                for j in range(n):
                        if G[i][j]==0:
                                if est_exposee(G,i,j) and
    bernouilli(p2)==1:
                                        Gprime[i][j]=1
                        if G[i][j]==1:
                                if bernouilli(p1)==1:
                                        Gprime[i][j]=3
                                else:
                                        Gprime[i][j]=2
                        else:
                                Gprime[i][j]=G[i][j]
        return Gprime
```

❼ On propose d'abord de définir une fonction auxiliaire vérifiant si la grille peut encore évoluer, c'est à dire s'il reste au moins une case infectée qui va de toute façon évoluer vers la guérison ou le décès:

Listing 30:

```
def est_infectee(G):
        n=len(G)
        bool=False
        for i in range(n):
                for j in range(n):
                        if G[i,j]==1:
                                bool=True
        return bool
```

puis la fonction simulation:

Listing 31:

```
def simulation(n,p1,p2):
        G=init(n)
        while est_infectee(G):
                G=suivant(Gancien,p1,p2)
        return [compte(G)[i]/n**2 for i in range(4)]
```

❽ La proportion x_1 des cases infectées en fin de simulation vaut nécessairement 0 puisqu'une case infectée ne peut pas rester indéfiniment dans cet état. Le test d'évolution de la grille dans la fonction `simulation` (à l'aide de la fonction auxiliaire `est_infectee`) s'appuie sur ce principe.

La somme des proportions des 4 états vaut naturellement $x0 + x1 + x2 + x3 = 1$. En fin de simulation, les cases ayant été touchées par la maladie se trouvent soit dans l'état `rétabli`, soit dans l'état `décédé`. Ainsi, cette proportion s'écrit $x\_atteinte = x2 + x3$

❾ On propose le code suivant pour la fonction `seuil(Lp2,Lxa)`:

Listing 32:

```
def seuil(Lp2,Lxa):
        g,d=0,len(Lxa)-1
        p2cmin,p2cmax=0.0,1.0
        while (d-g)>1:
                m=(g+d)//2
                if Lxa[m]==0.5:
                        return [Lp2[m],Lp2[m]]
                elif Lxa[m]<=0.5:
                        g=m
                        p2cmin=Lp2[m]
                else:
                        d=m
                        p2cmax=Lp2[m]
        return [p2cmin,p2cmax]
```