TD ipt^2 n° 2: Révisions 2/2

Preuve et complexité des algorithmes - extraits de concours

Algorithmes - Preuve d'algorithmes

Exercice n°1: Complexité et preuve d'un algorithme de calcul de série

On considère la série s_n des nombres rationnels définie par:

$$s_n = \sum_{k=1}^{n} \frac{1}{k} = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$$

❶ Rédiger un script Python qui réalise les opérations suivantes:

- Demander la saisie d'un flottant x.
- Déterminer le premier entier N tel que $s_N > x$ et affiche le résultat

On testera avec ce code avec les entiers $5, 10, \ldots$ puis avec le flottant $x = 17,4$

❷ **Analyse de scripts**

a· Démontrer par récurrence sur n que chacun des deux algorithmes qui suivent terminent et donneraient bien le résultat attendu si on pouvait calculer en précision infinie, sachant que dans le cours de mathématiques, on montre que:

$$\lim_{n\to+\infty} s_n = +\infty$$

b· Déterminer pour chacun d'eux une valeur de l'entier n au delà de laquelle la valeur du flottant s est constante. Quelle remarque peut-on faire?

c· Calculer avec précision, le nombre d'additions, de divisions qu'entraîne la réalisation de chacun d'eux en fonction de N la valeur affichée dans les cas où ils terminent.

Listing 1:

```
1 x=float(input('entrer␣x='))
2 s=1
3 n=1
4 while s<=x:
5     n+=1
6     s+=1/n
7 print ('avec␣N=',n,'s=',s)
```

Listing 2:

```
1 x=float(input('entrer␣x='))
2 s=1
3 n=1
4 while s<=x:
5     n+=1
6     s=1
7     for j in range(2,n+1):
8         s+=1/j
9 print ('avec␣N=',n,'s=',s)
```

❸ Le programme qui suit fait-il autre chose que les précédents? En quoi est-il préférable?

Listing 3:

```
1 import numpy as np
2 x=float(input('entrer␣x='))
3 n=1
4 while np.sum(np.array(range(1,n+1))**(-1.))<x:
5     n+=1
6 print ('avec␣N=',n,'s=',np.sum(np.array(range(1,n+1))**(-1.)))
```

Exercice n°2: Tri naïf

❶ Ecrire une fonction `max(L,deb)` d'argument `L` une liste et `deb` un entier qui renvoie la position du maximun et sa valeur dans la sous liste de `L` qui commence à `L[deb]`.

❷ On donne le code suivant.

Listing 4:

```
1 def trinaif(L):
2     n = len(L)
3     for k in range(n-1):
4         p, m = max(L, k)
5         L[k], L[p] = L[p], L[k]
6     return None
```

a· Expliquer la dernière instruction de la fonction.

b· Montrer que cet algorithme termine.

c· Soit L_j le contenu de L à l'issue de la $j^{\text{ième}}$ itération.
On appelle $\mathcal{P}_j$ la propriété :

$$\begin{cases} L_j[0] \geqslant L_j[1] \geqslant \cdots \geqslant L_j[j-1] \\ \forall i \geqslant j \quad L_j[i] \leqslant L_j[j-1] \end{cases}$$

Montrer que $(\mathcal{P}_j)$ est un invariant de boucle.

d· Calculer le nombre d'itérations de cet algorithme, et en déduire la complexité de celui-ci.

Exercice n°3: Recherche dichotomique

Dans cet exercice, on considère un tableau dont les éléments sont rangés dans l'ordre croissant. On cherche un algorithme qui permet de savoir si un élément est dans le tableau.

❶ Ecrire une fonction `app(e,T)` d'argument `e` et `T` et qui renvoie `True` (resp. `False`) si $e \in T$ (resp. $e \notin T$), sans tenir compte du fait que les éléments sont rangés.

❷ Calculer le nombre maximum d'itérations.

❸ On considère la stratégie suivante.

- On coupe le tableau en deux et on compare la valeur centrale du tableau et `e`. On en déduit le sous tableau dans lequel est `e`.
- On recommence l'opération précédente avec le sous tableau identifié à l'itération précédente.
- ...

Listing 5:

```
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:
            ...
        else:
            ...
    return ...
```

a· Compléter la fonction ci-dessus pour qu'elle réalise le but recherché.

b· On désigne par g_k et d_k les contenus de g et d après k itérations et n est la longueur du tableau.
Prouver que pour tout entier naturel k, $d_k - g_k < \frac{n}{2^k}$.

c· En déduire le nombre maximum d'itérations.

d· Prouver la terminaison de l'algorithme.

e· On suppose que $e \in T$. Prouver que $\forall k \in \mathbb{N}$, on entre dans la $k^{\text{ième}}$ itération avec $\mathcal{P}_k$: $g_k \leqslant d_k$ et $T[g_k] \leqslant e \leqslant T[d_k]$. En déduire la correction de l'algorithme.

Extraits de concours

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

Pour étudier l'effet du champ magnétique sur un matériau magnétique, on adopte une modélisation microscopique. On modélise les atomes par des sites portant chacun une grandeur physique, nommée *spin*, dont il n'est pas nécessaire de connaître les propriétés.

L'échantillon modélisé est une zone carrée à deux dimensions possédant h spins régulièrement répartis dans chaque direction, donc formant une grille carrée de $n = h^2$ *spins*. Chaque spin ne possède que deux états *down* ou *up*, ce que l'on modélise par une variable $s_i \in \{-1, +1\}$.

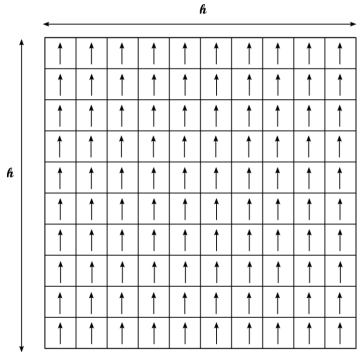

Figure 1: Modèle des spins dans un matériau ferromagnétique

Pour implémenter cette configuration de *spins* décrivant l'état microscopique du matériau, on choisit de travailler sur une liste s, contenant n entiers, chacun valant -1 ou 1. On notera le choix d'implémentation adoptée, qui impose de travailler sur une simple liste de n éléments pour modéliser une grille de taille $n = h \times h$, dans l'ordre suivant : première ligne puis deuxième ligne , etc.

Un domaine d'aimantation uniforme (cf. figure 1) sera donc représenté par une liste contenant n fois 1 ([1, 1, 1, ...1]).

❶ Écrire les instructions nécessaires pour importer exclusivement les fonctions exponentielle (exp) et tangente hyperbolique (tanh) du module math, ainsi que les fonctions `randrange` et `random` du module random.
Le début du programme (outre les imports réalisés ci-dessus) est défini par:

Listing 6:

```
h = 100
n = h**2
```

ce qui définit deux variables globales utilisables dans tout le programme.

❷ Ecrire une fonction initialisation() renvoyant une liste d'initialisation des domaines contenant n spins de valeur 1 comme sur la figure 1.

L'antiferromagnétisme est une propriété de certains milieux magnétiques. Contrairement aux matériaux ferromagnétiques, dans les matériaux antiferromagnétiques, l'interaction d'échange entre les atomes voisins conduit à un alignement antiparallèle des moments magnétiques atomiques (cf. figure 2). L'aimantation totale du matériau est alors nulle (on se limite au cas où h est pair).

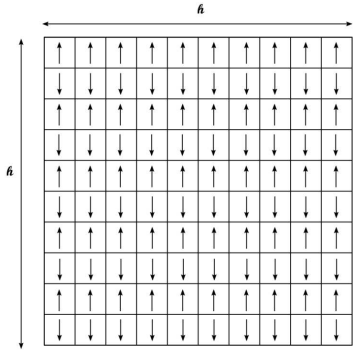

Figure 2: Modèle de spins d'un matériau antiferromagnétique

❸ Écrire une fonction `initialisation_anti()` renvoyant une liste s d'initialisation des domaines contenant h spins en largeur et h en hauteur en alternant les 1 et -1 comme sur la figure 2.

❹ Pour afficher l'état global du matériau, il est nécessaire de convertir la liste s utilisée en un tableau de taille $h \cdot h$ représenté par une liste de listes. Écrire une fonction `repliement(s)` qui prend en argument la liste de spins s et qui renvoie une liste de h listes de taille h représentant le domaine.

Attention: dans la suite, l'utilisation de la fonction repliement n'est pas autorisée: on travaille exclusivement sur une liste unidimensionnelle s.

Dans la modélisation adoptée (sans champ magnétique extérieur), l'énergie d'une configuration, définie par l'ensemble des valeurs de tous les spins, est donnée par:

$$E = -\frac{J}{2}\sum_i \sum_{j\in V_i} s_i s_j \tag{1}$$

avec V_i l'ensemble des voisins du spin i.

On suppose que seuls les quatre *spins* situés juste au dessus, en dessous, à gauche et à droite de s_i sont capables d'interagir avec lui.

J est nommée intégrale d'échange et modélise l'interaction entre deux spins voisins. Pour simplifier, on considérera dans les programmes que $J = 1$. Malgré le caractère fini de l'échantillon, on peut utiliser une modélisation très utile pour faire comme s'il était infini en utilisant les conditions aux limites périodiques. Lorsque l'on considère un spin dans la colonne située la plus à droite (resp. gauche), il ne possède pas de plus proche voisin à droite (resp. gauche) : on convient de lui en affecter un, qui sera situé sur la même ligne complètement à gauche (resp. droite) de l'échantillon . De même, le plus proche voisin manquant d'un spin situé sur la première (resp. dernière) ligne sera situé sur la dernière (resp. première) ligne de l'échantillon (voir la figure 3)

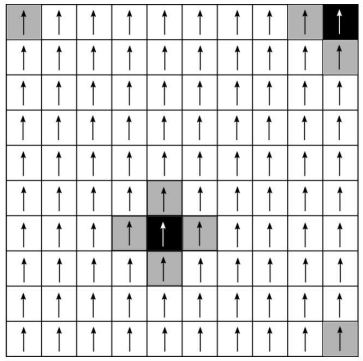

Figure 3: Voisinage d'un spin (les voisins des spins sur les cases noires sont indiqués en gris)

❺ Définir une fonction `liste_voisins(i)` qui renvoie la liste des indices des plus proches voisins du spin s_i d'indice i dans la liste s (dans l'ordre gauche, droite, dessous, dessus). On pourra utilement utiliser les opérations % et // de Python, qui renvoient le reste et le quotient de la division euclidienne.

❻ Définir la fonction `energie(s)` qui calcule l'énergie d'une configuration s donnée à partir de l'équation définissant E donnée plus haut.

Pour trouver une configuration stable pour les spins, il faut faire évoluer la liste vers une situation d'équilibre conformément aux principes de la physique statistique. On

adopte une méthode probabiliste de type méthode de Monte-Carlo, dont le principe de fonctionnement est le suivant. À chaque étape:

- On choisit un *spin* au hasard dans l'échantillon,
- on calcule la variation d'énergie ΔE qui résulterait d'un changement d'orientation de ce spin,
- si $\Delta E \leq 0$, ce *spin* change de signe,
- si $\Delta E > 0$, ce *spin* change de signe avec la probabilité $p(\Delta E)$ donnée par la loi de Boltzmann:

$$p = \exp\left(\frac{-\Delta E}{k_B T}\right)$$

Dans la suite, ΔE et $k_B T$ seront désignées par les variables `delta_e` et T correspondantes dans le programme.

❼ Définir une fonction `test_boltzmann(delta_e, T)` qui renvoie `True` si le spin change de signe, et `False` sinon

❽ Juste après avoir sélectionné au hasard l'indice i d'un spin de la liste s à basculer éventuellement, pour évaluer l'écart d'énergie `delta_e` entre les deux configurations avant/après, on propose deux solutions sous forme des fonctions `calcul_delta_e1` et `calcul_delta_e2` :

Listing 7:

```
def calcul_delta_e1(s, i):
    s2 = s[:]
    s2[i] = -s[i]
    delta_e = energie(s2)-energie(s)
    return delta_e
```

Listing 8:

```
def calcul_delta_e2(s, i):
    delta_e = 0
    for j in liste_voisins(i):
        delta_e = delta_e + 2*s[i]*s[j]
    return delta_e
```

où s[i] est le spin choisi pour être éventuellement retourné. Indiquer la solution qui vous paraît la plus efficace pour minimiser le temps de calcul en justifiant votre réponse.

❾ En utilisant la fonction `test_boltzmann`, définir une fonction `monte_carlo(s, T, n_tests)` qui applique la méthode de Monte-Carlo et qui modifie la liste s où l'on a choisi successivement `n_test` *spins* au hasard, que l'on modifie éventuellement suivant les règles indiquées dans les explications.

❿ Ecrire la fonction `aimantation_moyenne(n_tests, T)` qui :

- initialise une liste des *spins* (avec la fonction initialisation par exemple),
- la fait évoluer en effectuant `n_tests` tests de Boltzmann et les inversions éventuelles qui en découlent,
- calcule et renvoie l'aimantation moyenne de la configuration à la température T (définie ici comme la somme des valeurs des *spins* divisée par le nombre total de *spins*).

Annexe: Documentation sommaire

On donne les fonctions utiles suivantes:

- `randrange(n)` permet de renvoyer un entier aléatoirement choisi parmi $0, 1, 2, \ldots, n-1$.
- `random()` permet de renvoyer un flottant aléatoire entre 0 et 1 suivant une densité de probabilité uniforme.

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

Une bonne compréhension des mécanismes de déplacement des dunes est aujourd'hui un enjeu écologique important puisqu'elles constituent la barrière la moins couteuse contre le recul du trait de côte. Cela permet également de comprendre comment les déserts dévorent peu à peu certaines régions du globe (ils occupent à ce jour 16 millions de km^2 et ne cessent de s'étendre).

L'exercice qui suit, tiré de l'épreuve "0" du CCMP (2014) propose une modélisation du déplacement d'un tas de sable, constitué de quelques grains, par un automate cellulaire.

Un automate cellulaire se présente généralement sous la forme d'un quadrillage dont chaque case peut être occupée ou vide. Un grain y est symbolisé par une case occupée. La configuration des cases, qu'on appelle état de l'automate, évolue au cours du temps selon certaines règles très simples permettant de reproduire des comportements extrêmement complexes. La physique n'intervient pas directement mais les règles d'évolution sont choisies de façon à reproduire au mieux les lois naturelles.

Dans ce qui suit, nous allons simuler la formation d'un demi-tas de sable situé à droite de l'axe de symétrie vertical en appliquant les règles énoncées ci-après.

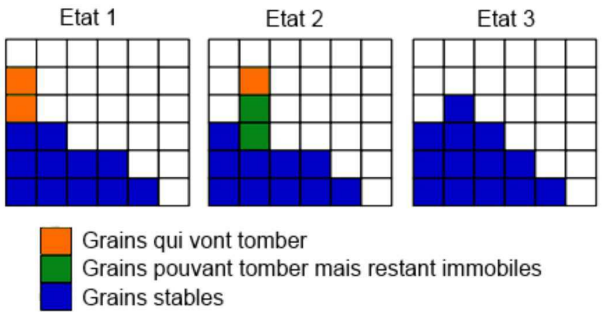

Figure 4: Exemple d'évolution de l'automate cellulaire à implémenter

Une tour de hauteur h est une pile de cases pleines consécutives dont h voisines de droite sont vides. Si $h > 1$, on détermine arbitrairement le nombre n de grains du sommet de la tour qui vont tomber avec l'instruction `Python` (après importation du module `random`):

```
n = int((h+2.0)/2 *random.random()) + 1
```

Dans l'exemple de la figure ci-dessus, lors du passage de l'état 1 à l'état 2, le hasard introduit par la fonction `random()` (qui renvoie de manière aléatoire un flottant dans l'intervalle semi-ouvert [0.0; 1.0[) fait que toute la tour se décale apparemment vers la droite. Ensuite, pour le passage à l'état 3, seul un grain sur les trois possibles tombe. L'état 3 est stable, plus aucun grain ne tombe.

❶ Déterminer un encadrement de n pour $h > 1$.

❷ Ecrire une fonction nommée `calcul_n` qui prend la hauteur h comme argument et qui renvoie le nombre n de grains qui vont tomber sur la pile suivante.

La représentation graphique est une image en deux dimensions mais l'automate est à une dimension car on considère le tas comme un ensemble de piles dont la hauteur future dépend uniquement des piles adjacentes. Le tas est complètement défini avec la variable `piles` qui permet de stocker la hauteur des différentes piles.

Au départ, le support est vide et on vient déposer périodiquement un grain sur la première pile (à gauche) qui correspondra au sommet du demi-tas. Le support peut recevoir P piles et à son extrémité droite il n'y a rien, les grains tombent et sont perdus. La pile $P + 1$ est donc toujours vide.

❸ Définir la fonction `initialisation(P)` renvoyant une variable `piles` de type liste contenant $P + 1$ piles de hauteur 0.

❹ Définir une fonction `actualise(piles,perdus)` qui va parcourir les piles de gauche à droite et les faire évoluer en utilisant les règles, fonctions, et variables définies précédemment. Cette fonction doit renvoyer la variable `piles` actualisée ainsi que le nombre de grains perdus (en prenant en compte ceux qui seront tombés de la dernière pile P).

❺ Écrire le bloc d'instructions du programme principal qui permet d'ajouter 1 grain à la première pile après chaque dizaine d'exécutions de la fonction `actualise(,)` et qui s'arrêtera lorsqu'au moins 1000 grains seront sortis du support. Le nombre de piles (taille du support P) sera demandé à l'utilisateur lors de l'exécution. Vous utiliserez les fonctions et variables définies précédemment.

❻ Comment tracer simplement la forme (allure) du demi-tas à la fin de la simulation précédente.

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

On s'intéresse dans cet exercice à une méthode de modélisation de la propagation d'une épidémie par un automate cellulaire.

Dans ce qui suit, on appelle grille de taille $n \times n$ une liste de n listes de longueur n, où n est un entier strictement positif.

Pour mieux prendre en compte la dépendance spatiale de la contagion, il est possible de simuler la propagation d'une épidémie à l'aide d'une grille (automate cellulaire). Chaque case de la grille peut être dans un des quatre états suivants : saine, infectée, rétablie, décédée. On choisit de représenter ces quatre états par les entiers:

0 (Sain), 1 (Infecté), 2 (Rétabli), et 3 (Décédé)

L'état des cases d'une grille évolue au cours du temps selon des règles simples. On considère un modèle où l'état d'une case à l'instant $t + 1$ ne dépend que de son état à l'instant t et de l'état de ses huit cases voisines à l'instant t (une case du bord n'a que cinq cases voisines et trois pour une case d'un coin). Les règles de transition sont les suivantes:

- une case décédée reste décédée.
- une case infectée devient décédée avec une probabilité p_1 ou rétablie avec une probabilité $(1 - p_1)$.
- une case rétablie reste rétablie.

- une case saine devient infectée avec une probabilité p_2 si elle a au moins une case voisine infectée et reste saine sinon.

On initialise toutes les cases dans l'état sain, sauf une case choisie au hasard dans l'état infecté.

❶ On a écrit en Python la fonction `grille(n)` suivante:

Listing 9:

```
1 def grille(n):
2   M=[]
3   for i in range(n):
4     L=[]
5     for j in range(n): L.append(0)
6     M.append(L)
7   return M
```

Décrire ce que retourne cette fonction.

On pourra dans la question suivante utiliser la fonction `randrange(p)` du module `random` qui, pour un entier positif p, renvoie un entier choisi aléatoirement entre 0 et $p = 1$ exclus.

❷ Ecrire en `Python` une fonction `init(n)` qui construit une grille G de taille $n \times n$ ne contenant que des cases saines, choisit aléatoirement une des cases et la transforme en case infectée, et enfin renvoie G.

❸ Ecrire en `Python` une fonction `compte(G)` qui a pour argument une grille G et renvoie la liste `[n0, n1, n2, n3]` formée des nombres de cases dans chacun des quatre états.

D'après les règles de transition, pour savoir si une case saine peut devenir infectée à l'instant suivant, il faut déterminer si elle est exposée à la maladie, c'est-à-dire si elle possède au moins une case infectée dans son voisinage. Pour cela, on écrit en Python la fonction `est_exposee(G, i, j)` suivante:

Listing 10:

```
1  def est_exposee(G, i, j):
2    n = len(G)
3    if i==0 and j==0:
4      return (G[0][1]-1)*(G[1][1]-1)*(G[1][0]-1)==0
5    elif i==0 and j==n-1:
6      return (G[0][n-2]-1)*(G[1][n-2]-1)*(G[1][n-1]-1)== 0
7    elif i==n-1 and j == 0:
8      return (G[n-1][1]-1)*(G[n-2][1]-1)*(G[n-2][0]-1)==0
9    elif i==n-1 and j==n-1:
10     return (G[n-1][n-2]-1)*(G[n-2][n-2]-1)*(G[n-2][n-1]-1)==0
11   elif i==0:
12     # a completer
13   elif i==n-1:
14     return (G[n-1][j-1]-1)*(G[n-2][j-1]-1)*(G[n-2][j]-1)*(G[n-2][j+1]-1)*(G[n-1][j+1]-1)==0
15   elif j==0:
16     return (G[i-1][0]-1)*(G[i-1][1]-1)*(G[i][1]-1)*(G[i+1][1]-1)*(G[i+1][0]-1)==0
17   elif j==n-1:
18     return (G[i-1][n-1]-1)*(G[i-1][n-2]-1)*(G[i][n-2]-1)*(G[i+1][n-2]-1)*(G[i+1][n-1]-1)==0
19   else:
20     # a completer
```

❹ Quel est le type du résultat renvoyé par la fonction `est_exposee`?

❺ Compléter les lignes 12 et 20 de la fonction `est_exposee`.

❻ Ecrire une fonction `suivant(G, p1, p2)` qui fait évoluer toutes les cases de la grille G à l'aide des règles de transition et renvoie une nouvelle grille correspondant à l'instant suivant. Les arguments p_1 et p_2 sont les probabilités qui interviennent dans les règles de transition pour les cases infectées et les cases saines. On pourra utiliser la fonction `bernoulli(p)` suivante qui simule une variable aléatoire de Bernoulli de paramètre p : `bernoulli(p)` vaut 1 avec la probabilité p et 0 avec la probabilité (1?p):

Listing 11:

```
1 def bernouilli(p):
2   x=rd.random()
3   if x<=p:
4     return 1
5   else:
6     return 0
```

On reproduit ci-dessous le descriptif de la documentation Python concernant la fonction `random` de la bibliothèque `random`:

> `random.random()`
> `Return the next random floating point in the range` [0.0, 1.0[

Avec les règles de transition du modèle utilisé, l'état de la grille évolue entre les instants t et $t + 1$ tant qu'il existe au moins une case infectée.

❼ Ecrire en `Python` une fonction `simulation(n, p1, p2)` qui réalise une simulation complète avec une grille de taille $n \times n$ pour les probabilités $p1$ et $p2$, et renvoie la liste $[x0, x1, x2, x3]$ formée des proportions de cases dans chacun des quatre états à la fin de la simulation (une simulation s'arrête lorsque la grille n´évolue plus).

❽ Quelle est la valeur de la proportion des cases infectées $x1$ à la fin d?une simulation ? Quelle relation vérifient $x0$, $x1$, $x2$ et $x3$? Comment obtenir à l'aide des valeurs de $x0$, $x1$, $x2$ et $x3$ la valeur `x_atteinte` de la proportion des cases qui ont été atteintes par la maladie pendant une simulation ?

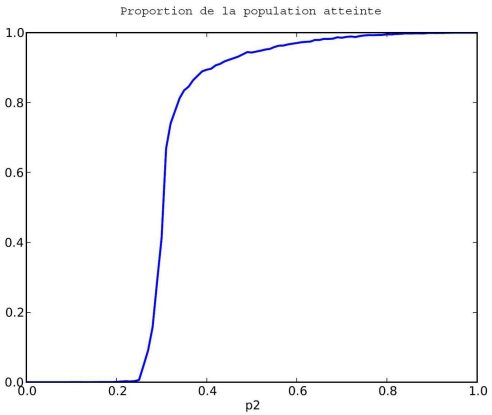

Figure 5: Représentation de la proportion de la population qui a été atteinte par la maladie pendant la simulation en fonction de la probabilité $p2$

On fixe p1 à 0.5 et on calcule la moyenne des résultats de plusieurs simulations pour différentes valeurs de $p2$. On obtient la courbe de la figure 2.

❾ On appelle seuil critique de pandémie la valeur de $p2$ à partir de laquelle plus de la moitié de la population a été atteinte par la maladie à la fin de la simulation. On suppose que les valeurs de $p2$ et `x_atteinte` utilisées pour tracer la courbe de la figure 2 ont été stockées dans deux listes de même longueur `Lp2` et `Lxa`. Ecrire en `Python` une fonction `seuil(Lp2, Lxa)` qui détermine par dichotomie un encadrement $[p2cmin, p2cmax]$ du seuil critique de pandémie avec la plus grande précision possible. On supposera que la liste `Lp2` croît de 0 à 1 et que la liste `Lxa` des valeurs correspondantes est croissante.