

## TD IPT<sup>2</sup> N° 2: RÉVISIONS 2/2

### REPRÉSENTATION DES NOMBRES - PREUVE ET COMPLEXITÉ DES ALGORITHMES

#### Représentation des nombres

##### EXERCICE N°1: Conversion

On considère la fonction `mystere` suivante:

LISTING 1:

```
1 def mystere(n):
2     res = 0
3     aux = 1
4     while (n != 0):
5         b = n % 10
6         if b < 2:
7             res = res + aux*b
8             aux = aux*2
9             n = n // 10
10        else:
11            print("Erreur")
12            return None
13    return res
```

- 1 Tester "à la main" cette fonction pour les entiers  $n=301$  puis  $n=1010$  en donnant à chaque itération les différentes valeurs prises par les variables `b`, `res`, `aux`, et `n`.
- 2 Que fait finalement la fonction `mystere`?

##### EXERCICE N°2: Différence entre réels et flottants

On considère la série numérique suivante:

$$S_n = \sum_{k=1}^n \frac{(-1)^{k-1}}{k}$$

On démontre, et nous admettons ici ces résultats que:

$$\begin{cases} \text{pour tous } m, n \in \mathbb{N}^*, \text{ si } m \leq n, S_{2m} \leq S_{2n} \leq \ln(2) \leq S_{2n+1} \leq S_{2m+1} \\ \text{pour tout } n \in \mathbb{N}^*, |S_n - \ln 2| \leq \frac{1}{n+1} \end{cases}$$

- 1 Montrer que:


$$\sum_{k=1}^{2n} \frac{(-1)^{k-1}}{k} = \frac{1}{2} \sum_{k=1}^n \frac{1}{(2k-1)k}$$


- 2 Ecrire les deux fonctions Python `S1(n)` et `S2(n)` correspondant aux deux formulations précédentes de la série  $S_n$ .
- 3 Calculer le nombre maximal de chiffres décimaux significatifs pour les flottants dans la norme IEEE754 (la mantisse étant codée sur 52+1 bits).
- 4 On donne le script Python suivant:

LISTING 2:

```
1 import time as t
2 import numpy as np
3 n = int(input("n="))
4
5 for p in range(n, n+2):
6     t0 = t.time()
7     print(S1(2*p))
8     print(S2(p))
9     print(S1(2*p+1))
10    print(np.log(2))
11    print("temps d'exécution t=", t.time()-t0)
12    print("-----")
```

Ce script donne les résultats suivants:


n=1000
0.69289724306
0.69289724306
0.693396993185
0.69314718056
temps exécution t= 0.00399994850159
0.692897492685
0.692897492685
0.693396743809
0.69314718056
temps exécution t= 0.00600004196167


n=10000000
0.69314715556
0.693147155562
0.69314720556
0.69314718056
temps exécution t= 36.3599998951
0.69314715556
0.693147155562
0.69314720556
0.69314718056
temps exécution t= 36.620000124

Les résultats obtenus sont-ils conformes à ce que l'on sait des propriétés de cette suite?  
QUESTION PRATIQUE: entrer ce script (et également celui des fonctions S1(n) et S2(n)) et lancer le pour les valeurs  $n = 1000$  et  $n = 10000000$ .

### EXERCICE N°3: De l'usage du schéma de Horner

On rappelle l'écriture du schéma de Horner permettant l'évaluation d'un polynôme  $P_n(x)$  de degré  $n$  et de coefficients  $\{a_0, a_1, a_2, \dots, a_n\}$

$$P_n(x) = a_0 + a_1 \cdot x + a_2 \cdot x^2 + \dots + a_{n-1} \cdot x^{n-1} + a_n \cdot x^n$$

s'écrit:

$$P_n(x_0) = (((a_n \cdot x_0 + a_{n-1}) \cdot x_0 + a_{n-2}) \cdot x_0 + a_{n-3}) \dots x_0 + a_0$$

- 1 Rappel le principe de conversion d'un nombre écrit en base  $x$  en base 10.
- 2 Exploiter le schéma de Horner pour réaliser la conversion d'un nombre hexadécimal en décimal. On pourra introduire la liste des digits de la base hexadécimale  $[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, "a", "b", "c", "d", "e", "f"]$ .
- 3 Expliquer l'intérêt de procéder ainsi.

### EXERCICE N°4: Base hexadécimale

On considère la base dite "hexadécimale".

Pour écrire un nombre dans cette base, il faut exploiter les chiffres de 0 à 9, puis les lettres a,b,c,d,e,f.

- 1 Convertir l'entier hexadécimal  $(a12f)_{16}$  en base décimale.
- 2 Que fait le script python suivant:

LISTING 3: Que fais-je?

```
1 def Chiffre (n) :
2     """n est un entier naturel donné par son écriture décimale"""
3     c=[]
4     while n!=0 :
5         c.append(n%16)
6         n//=16
7     c.reverse() # inverse l'ordre des éléments d'une liste
8     return c
9 print Chiffre (41263)
```

Ce programme présente un inconvénient. Lequel. Proposer un nouveau script corrigeant ce défaut. On pourra par exemple constituer une liste des "digits" de la base hexadécimale, soit  $['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c', 'd', 'e', 'f']$ , dont les indices vont naturellement de 0 à 15 pour résoudre le souci de notre programme.

### EXERCICE N°5: Fractions égyptiennes

Une fraction égyptienne est l'écriture d'un nombre rationnel sous la forme d'une somme de fractions unitaires (dont le numérateur est égal à 1) telles que les dénominateurs soient deux à deux distincts, c'est à dire sous la forme:

$$\frac{p}{q} = \sum_i \frac{1}{b_i}, \quad \text{avec } \forall i, j \neq i, b_i \neq b_j, p, q \in \mathbb{Z} \times \mathbb{Z}^*, b_i \in \mathbb{Z}^*$$

Par exemple, on peut écrire:

$$\frac{4}{5} = \frac{1}{2} + \frac{1}{4} + \frac{1}{20}$$

Ecrire un script Python qui permet une décomposition d'un rationnel en fractions égyptiennes. On calculera cette décomposition à l'aide du résultat suivant:

$$\frac{x}{y} = \frac{1}{\lceil y/x \rceil} + \frac{-y \% x}{y \lceil y/x \rceil}$$

que l'on appliquera de manière itérative sur la fraction non unitaire.

## Algorithmes - Preuve d'algorithmes

### EXERCICE N°6: Complexité et preuve d'un algorithme de calcul de série

On considère la série  $s_n$  des nombres rationnels définie par:

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

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, ... 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:

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

- 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 4:

```
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 5:

```
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 6:

```
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°7: Jeu de Nim

On dispose de  $N$  jetons identiques disposés sur une table. Deux joueurs  $A$  et  $B$  jouent à tour de rôle. Chacun doit, lorsque c'est son tour, retirer soit 1, 2 ou 3 jetons. Le joueur qui retire le dernier jeton a perdu. On dispose des codes suivants.

LISTING 7:

```

1 def nim(n):
2     N = n
3     while N != 0:
4         if N==1:
5             print ('B_gagne')
6             break
7         N = A_joue(N)
8         if N == 0:
9             r = 'B_gagne'
10        else:
11            if N==1:
12                print ('A_gagne')
13                break
14            N = B_joue(N)
15            if N == 0:
16                r = 'A_gagne'
17        return r
18
19 def A_joue(N):
20     r = divmod(N,4) [1]
21     if r == 0:
22         return N-3
23     elif r == 2:
24         return N-1
25     elif r == 3:
26         return N-2
27     else:
28         return N - randint(1,3)

```

- ❶ Montrer que l'algorithme se termine.
- ❷ On suppose dans cette question que  $n \not\equiv 1 \pmod 4$ .
  - a. On désigne par  $N_k$  le contenu de  $N$  après  $k$  itérations. Prouver que «  $N_k \not\equiv 1 \pmod 4$  » est un invariant de boucle.
  - b. En déduire que  $A$  gagne, quelque soit la fonction  $B\_joue$ .
- ❸ Quelle stratégie doit adopter  $B$  pour être sûr de gagner la partie quelque soit le résultat de la fonction  $A\_joue$ ?

### EXERCICE N°8: Tri naïf

- ❶ Ecrire une fonction  $\text{max}(L, \text{deb})$  d'argument  $L$  une liste et  $\text{deb}$  un entier qui renvoie la position du maximum et sa valeur dans la sous liste de  $L$  qui commence à  $L[\text{deb}]$ .
- ❷ On donne le code suivant.

LISTING 8:

```

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

$$\begin{cases} L_j[0] \geq L_j[1] \geq \dots \geq L_j[j-1] \\ \forall i \geq j \quad L_j[i] \leq L_j[j-1] \end{cases}$$

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°9: 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  $\text{app}(e, T)$  d'argument  $e$  et  $T$  et qui renvoie  $\text{True}$  (resp.  $\text{False}$ ) 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 9:

```

1 def dichotomie(e, T):
2     g, d = 0, len(T)-1
3     while g <= d:
4         m = (g + d) // 2
5         if T[m] == e:
6             return True
7         if T[m] < e:
8             ...
9         else:
10            ...
11    return ...

```

- Compléter la fonction ci-dessus pour qu'elle réalise le but recherché.
- 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  $k$ ,  $d_k - g_k < \frac{n}{2^k}$ .
- En déduire le nombre maximum d'itérations.
- Prouver la terminaison de l'algorithme.
- On suppose que  $e \in T$ . Prouver que  $\forall k \in \mathbb{N}$ , on entre dans la  $k^{\text{ième}}$  itération avec  $\mathcal{P}_k : g_k \leq d_k$  et  $T[g_k] \leq e \leq T[d_k]$ . En déduire la correction de l'algorithme.

### EXERCICE N°10: Fusion ordonnée de deux tableaux

L'objet de cet exercice est de construire un tableau d'éléments ordonnés dans l'ordre croissant à partir de deux tableaux également ordonnés dans l'ordre croissant.

- On peut ranger les éléments d'un tableau de taille  $n$  dans l'ordre croissant de la manière suivante :
  - On parcourt le tableau et on place le max en dernière position.
  - On parcourt le sous tableau des  $n-1$  premiers éléments et on place le max en dernière position du sous tableau.
  - ...

Compléter le code suivant où la fonction `tri_select` d'argument un tableau  $T$  renvoie le tableau trié dans l'ordre croissant. à l'aide de cette fonction, écrire une fonction

`class(T,U)` d'argument  $T$  et  $U$  des tableaux et qui renvoie le tableau trié fabriqué à partir de  $T$  et  $U$ .

LISTING 10:

```

1 def tri_select(T):
2     n = len(T)
3     for k in range(n-1, 0, -1):
4         max = T[0]
5         rg_max = 0
6         for j in range(...):
7             if T[j] > max:
8                 .....
9                 .....
10            if T[k] < max:
11                .....
12                .....
13    return T

```

- Voici une manière directe d'imbriquer deux tableaux triés.

- On compare  $T[0]$  et  $U[0]$  et on met le min dans  $R[0]$ .
- Si  $T[0] < U[0]$ , on compare  $T[1]$  et  $U[0]$  et on met le min dans  $R[1]$ .
- ... Il ne faut pas oublier que les tableaux ne sont pas forcément de même taille.

Compléter le code suivant où la fonction d'arguments les deux tableaux  $T$  et  $U$ , renvoie le tableau trié fabriqué à partir de  $T$  et  $U$ . Le module `numpy` a été renommé `np`.

LISTING 11:

```

1 def interclassement(T,U):
2     t, u, n = len(T), len(U), t+u
3     R = np.array(np.ones(n))
4     i, j, k = 0, 0, 0
5     while (i < t) and (j < u):
6         if T[i] < U[j]:
7             .....
8             i, k = .....
9         else:
10            .....
11    if i == t:
12        while j < u:
13            .....
14    else:
15        .....
16    return R

```

Morceaux choisis d'épreuves

### EXERCICE N°11:

### 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.

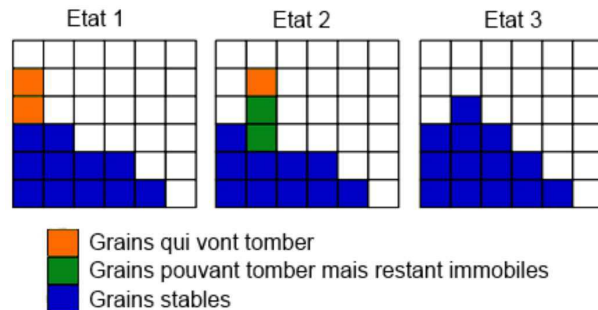


FIGURE 1: Exemple d'évolution de l'automate cellulaire à implémenter

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.

- 1 Déterminer un encadrement de  $n$  pour  $h > 1$ .
- 2 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.

- 3 Définir la fonction `initialisation(P)` renvoyant une variable `piles` de type liste contenant  $P + 1$  piles de hauteur 0.
- 4 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  $P$ ).
- 5 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.
- 6 Comment tracer simplement la forme (allure) du demi-tas à la fin de la simulation précédente.

### EXERCICE N°12:

### 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.

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

LISTING 12:

```
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.

- 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  $G$ .
- 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 13:

```
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][
15        j+1]-1)*(G[n-1][j+1]-1)==0
16    elif j==0:
17        return (G[i-1][0]-1)*(G[i-1][1]-1)*(G[i][1]-1)*(G[i+1][1]-1)
18    *(G[i+1][0]-1)==0
19    elif j==n-1:
20        return (G[i-1][n-1]-1)*(G[i-1][n-2]-1)*(G[i][n-2]-1)*(G[i+1][
21    n-2]-1)*(G[i+1][n-1]-1)==0
22    else:
23        # a completer
```

- Quel est le type du résultat renvoyé par la fonction `est_exposee`?
- Compléter les lignes 12 et 20 de la fonction `est_exposee`.
- 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 14:

```
1 def bernoulli(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.

- ⑦ 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  $p_1$  et  $p_2$ , et renvoie la liste  $[x_0, x_1, x_2, x_3]$  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).
- ⑧ Quelle est la valeur de la proportion des cases infectées  $x_1$  à la fin d'une simulation ? Quelle relation vérifient  $x_0, x_1, x_2$  et  $x_3$  ? Comment obtenir à l'aide des valeurs de  $x_0, x_1, x_2$  et  $x_3$  la valeur `x_atteinte` de la proportion des cases qui ont été atteintes par la maladie pendant une simulation ?

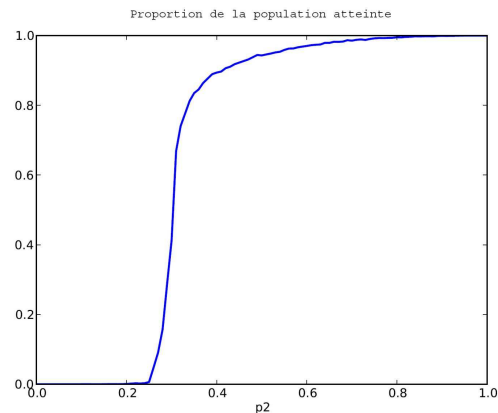


FIGURE 2: Représentation de la proportion de la population qui a été atteinte par la maladie pendant la simulation en fonction de la probabilité  $p_2$

On fixe  $p_1$  à 0.5 et on calcule la moyenne des résultats de plusieurs simulations pour différentes valeurs de  $p_2$ . On obtient la courbe de la figure 2.

- ⑨ On appelle seuil critique de pandémie la valeur de  $p_2$  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  $p_2$  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  $[p_{2cmin}, p_{2cmax}]$  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.