Correction du TD ipt^2 n° 2: Révisions 2/2
Représentation des nombres - preuve et complexité des algorithmes

---

Exercice n°1: **Conversion**

❶ Tableau d'évolution des variables:

- **Pour la valeur** $n = 301$**:**

rang	b	res	aux	n
0	non défini	0	1	301
1	1	1	2	30
2	0	1	4	3
3	$3 > 2$ affiche "erreur"	1	4	3

- **Pour la valeur** $n = 1010$**:**

rang	b	res	aux	n
0	non défini	0	1	1010
1	0	0	2	101
2	1	2	4	10
3	0	2	8	1
4	1	affiche 10	16	0

❷ La fonction mystère convertit donc les entiers binaires en entiers décimaux.

Exercice n°2: **Différence entre réels et flottants**

❶ La somme à calculer est une somme alternée:

$$\sum_{k=1}^{2n} \frac{(-1)^{k-1}}{k} = 1 - \frac{1}{2} + \frac{1}{3} + \ldots - \frac{1}{2n}$$

Elle peut donc également s'écrire en distinguant les cas k pair et k impair, soit:

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

❷ Propositions de scripts pour les fonctions S_1 et S_2:

Listing 1:

```python
def S1(n):
    res=0
    for k in range(1,2*n+1):
        res=res+(-1)**(k-1)/float(k)
    return res
```

Listing 2:

```python
def S2(n):
    res=0
    for k in range(1,n+1):
        res=res+1/(float(k)*(2*float(k)-1))
    return res/2
```

❸ Supposons un nombre décimal m tel que $10^n \leq m \leq 10^{n+1}$, alors m s'écrit en base 10 avec un nombre de chiffres significatifs $n+1$ tel que:

$$n = \lfloor \log_{10}(m) \rfloor = \lfloor \frac{\ln(m)}{\ln 10} \rfloor$$

NB: démarche déjà rencontrée dans l'exercice n°8 du TD1 sur la structure du code INSEE (test de bon formatage du code INSEE)

En base 2, la démarche est identique: si le nombre m décimal vérifie $2^p \leq m \leq 2^{p+1}$ alors le nombre de bits significatifs $p+1$ est tel que p vérifie:

$$p = \lfloor \log_2(m) \rfloor = \lfloor \frac{\ln(m)}{\ln(2)} \rfloor = 52\ (bits)$$

On a donc:

$$n = \log_{10}(m) = \lfloor \frac{\ln(m)}{\ln 10} \rfloor \simeq \lfloor \frac{\ln(m)}{\ln(2)} \rfloor \cdot \frac{\ln(2)}{\ln(10)} = 52 \cdot \frac{\ln(2)}{\ln(10)} \sim 15,65$$

soit finalement un nombre total de chiffres significatifs: $n+1 \simeq 16,65 \longrightarrow$ 16 chiffres significatifs

❹ On fera les commentaires sur la première exécution réalisée pour $n = 1000$ (les commentaires étant reconductibles pour le cas $n = 10^7$):

- On constate bien que l'encadrement proposé dans l'énoncé est respecté à savoir:

$$S_{2p} \leq \ln(2) \leq S_{(2p+1)}$$

(la présence dans le script de l'évaluation numérique de $S_2(p)$ vise simplement à montrer qu'une différence numérique entre $S_1(2p)$ et $S_2(p)$ apparaît mais à un rang p très important.)

- Par ailleurs, l'erreur maximale commise sur l'évaluation de ln(2) doit selon la seconde propriété énoncée, apparaître sur la $\log_{10}(n)+1$ ième décimale; c'est bien le cas ici puisque on constate que les deux nombres diffèrent à partir de la 3ième décimale.

Exercice n°3: De l'usage du schéma de Horner

❶ On écrit:

$$N_x = (q_n q_{n-1} \dots q_1 q_0)_x = q_n \cdot x^n + q_{n-1} \cdot x^{n-1} + q_{n-2} \cdot x^{n-2} \dots q_1 \cdot x^1 + q_0 = N_{10}$$

❷ On propose le code suivant:

Listing 3:

```
def convhex(N):
    Ldigits=['0','1','2','3','4','5','6','7','8','9','a','b','c','d','e','f']
    Lnbre=[]
    for car in N:
        if car not in Ldigits:
            print "erreur de saisie"
            return
        for p in range(len(Ldigits)):
            if car==Ldigits[p]:
                Lnbre.append(p)
    res=Lnbre[0]
    for k in Lnbre[1:]: #boucle de calcul type Horner
        res=res*16+k
    return res
```

Autre proposition de code qui calcule calcule le schéma de Horner au fur et à mesure de la conversion des digits:

Listing 4:

```
def convhex2(N):
        Ldigits=['0','1','2','3','4','5','6','7','8','9','a','b','c','d','e','f']
        Lnbre=[]
        res=0
        for k in N:
                for i in range(16):
                        if k==Ldigits[i]:
                                Lnbre.append(i)
                                if len(Lnbre)==1:
                                        res=Lnbre[0]
                                else:
                                        res=res*16+Lnbre[-1]
        return res
```

❸ L'utilisation du schéma de Horner permet comme toujours d'éviter les opérations d'exponentiation coûteuse en temps machine.

Exercice n°4: Base hexadécimale

❶ $(a12f)_{16} = 15\times16^0 + 2\times16^1 + 1\times16^2 + 10\times16^3 = 41263_{10}$

❷ c est initialement une liste vide. On traduit l'entier naturel n par division euclidienne successive du quotient, les différents restes constituant alors les digits du nombre traduit en hexadécimal. Ces digits sont ajoutés à chaque itération dans la liste c.

La commande reverse permet d'inverser l'ordre des éléments de la liste afin de représenter correctement le nombre hexadecimal avec ses digits de poids croissants vers la gauche.

Inconvénient du programme: pour les digits de valeur supérieure à 9, le programme écrit 10,11,12,13,14, ou 15 et non a,b,c,d,e, ou f. On propose le script suivant permettant de corriger le problème:

Listing 5:

```
    def Chiffre (n) :
    c=[]
    #Constitution de l'abcdaire
    abcdaire=[0,1,2,3,4,5,6,7,8,9,"a","b","c","d","e","f"]
    while n!=0 :
```

```python
            c.append(abcdaire[n%16])
            n//=16
        c.reverse() # inverse l'ordre des éléments d'une liste
        res=""
        for elt in c: res=res+str(elt)
        return res
print Chiffre (41263)
```

Exercice n°5: Fractions égyptiennes

On propose le script suivant:

Listing 6:

```python
import math
N=0.0
D=0.0
while not(type(N)==int) and not(type(D)==int):
        N=int(input("Entrez le numerateur: "))
        D=int(input("Entrez le denominateur: "))
res=str(int(N))+"/"+str(int(D))+"="
while -int(D)%int(N)!=0:
        res=res+"+"+"1/"+str(int(math.ceil(float(D)/float(N))))
        num=N
        N=(-D)%int(N)
        D=D*math.ceil(float(D)/float(num))
res=res+"+"+"1/"+str(int(math.ceil(D/N)))
print(res)
```

Algorithmes - Preuve d'algorithmes

Exercice n°6: 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°7: Jeu de Nim

1. - la boucle s'execute tant que $N! = 0$
   - On vérifie s'il reste $N = 1$ jeton avant que A ne joue. Si c'est le cas alors on affiche "B gagne" et on stoppe le programme qui se termine.
   - A joue et on calcule le reste de la division euclidienne de N par 4 que l'on stocke dans r
   - les 3 conditions testées sur la valeur de r conduisent à un décrément dans l'ordre des conditions de 3, 1, 2 ou un entier 1 ou 2 (`randint`) de la valeur de N qui est renvoyée dans le programme.
   - On vérifie s'il reste $N = 1$ jeton avant que B ne joue. Si c'est le cas alors on affiche "A gagne" et on stoppe le programme qui se termine.
   - B joue et on calcule le reste de la division euclidienne de N par 4 que l'on stocke dans r (en supposant la fonction `B_joue` identique à `A_joue`.
   - les 3 conditions testées sur la valeur de r conduisent à un décrément dans l'ordre des conditions de 3, 1, 2 ou un entier 1 ou 2 (`randint`) de la valeur de N qui est renvoyée dans le programme. Ainsi au rang $k+1$ d'itération on a $N_{k+1} < N_k$ et soit la condition de boucle finit par être violée i.e. $N = 0$, l'algorithme termine alors, ou bien on obtient le cas $N = 1$ qui termine également l'algorithme.

2. **a·** D'après l'hypothèse, avant la première itération on a: $N_0 = n \not\equiv 1[4]$.
   Supposons vraie la propriété au rang k avec: $N_k \not\equiv 1[4]$, soit pour $p \in \mathbb{N}$, on a 3 possibilités:

$$\begin{cases} N_k = 4p \implies N_{k'} = \texttt{A\_joue}(N_k) = 4p - 3 = 4(p-1) + 1 \\ N_k = 4p + 2 \implies N_{k'} = \texttt{A\_joue}(N_k) = 4p + 1 \\ N_k = 4p + 3 \implies N_{k'} = \texttt{A\_joue}(N_k) = 4p + 1 \end{cases}$$

   NB: La situation est identique lorsque B joue.

   Ainsi, au rang $N_{k'}$ on a $N_{k'} \equiv 1[4]$.
   puis B va jouer (on suppose que `B_joue` possède la même structure que `A_joue`); on aura donc:

$$N_{k+1} = \texttt{B\_joue}(N_{k'}) = N_{k'} - 1 \text{ ou } N_{k'} - 2 \text{ ou } N_{k'} - 3$$

   Bilan: au rang $k+1$ on a bien

$$N_{k'} \not\equiv 1[4]$$

   b· Lorsque B s'apprête à jouer, on a toujours $N_k = 1[4]$ qui est une situation perdante. Plus précisément, il existe forcément un rang d'itération, qui se produit pour $p = 0$ ou $p = 1$ pour lequel les valeurs possibles de N sont 2,3 ou 4 compte tenu de l'invariant de boucle. Ainsi, A n'aura qu'à retirer respectivement 1, 2 ou 3 jetons afin de laisser à B un seul jeton et gagner.

3. Pour gagner, B doit faire jouer A avec $4p+1$ jetons, soit 1[4]; ainsi en retirant à son tour 1,2 ou 3 jetons, A laissera pour B une situation de gagnant soit $N_B \not\equiv 1[4]$. B gagnera à coup sûr.

Exercice n°8: Tri naïf

Listing 7:

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

2. **a·** Dans ce script, l'argument L est une liste donc un objet mutable modifié au cours de l'exécution du programme. Le nouveau contenu de la liste est renvoyé en fin de fonction dans l'argument L. La commande `None` permet de terminer la fonction sans renvoi précis. Compte tenu du caractère mutable d'une liste: les modifications s'établissent directement à l'adresse mémoire pointant sur la liste.

   b· Le script exploite une boucle inconditionnelle `for` qui de fait se termine au bout d'un nombre d'itérations fixé.

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

$$\begin{cases} L_1[0] \leq L_1[0] \text{ vrai après 1ère itération car égalité} \\ L_1[i] \leq L_1[0] \forall i \geq 1 \text{ vrai après 1ère itération car max place en position 0 le maximum} \end{cases}$$

   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 $L[j]\ldots L[n-1]$ en $L_{j+1}[j]$ donc:

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

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 execute n itérations
- Pour la 2^{nde} itération $k=1$ de la première boucle, la boucle de max execute $n-1$ itérations
- Pour la $k+1^{\text{ième}}$ itération de la première boucle, la boucle de max execute $n-k$) itérations
- ...
- Pour la $n-1^{\text{ième}}$ itération $k=n-2$ de la première boucle, la boucle de max execute $n-(n-2)=2$ itérations

Finalement, le nombre total d'itérations est:

$$\sum_{k=0}^{n-2} n-k = (2+3+\ldots+n) = (n-1)\frac{n+2}{2}$$

Exercice n°9: 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 8:

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

Seconde proposition: avec boucle inconditionnelle

Listing 9:

```
    def app(e,T):
        for pos in range(len(T)):
            if T[pos]==e:
                return True
        return False
```

Troisième proposition: avec boucle conditionnelle

Listing 10:

```
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 11:

```
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 propose deux méthodes. La première, progressive (un peu trop pour certains!):

- **au rang** 1:

$$d_1-g_1 \overset{d_1=m-1}{=} \left(E\left[\frac{g_0+d_0}{2}\right]-1\right)-g_0 < \frac{d_0+g_0}{2}-g_0 = \frac{d_0-g_0}{2} < \frac{n}{2}$$

ou

$$d_1-g_1 \overset{g_1=m+1}{=} d_0-\left(E\left[\frac{g_0+d_0}{2}\right]+1\right) < d_0-\frac{g_0+d_0}{2} = \frac{d_0-g_0}{2} < \frac{n}{2}$$

- **au rang** 2:

$$d_2-g_2 \overset{d_2=m-1}{=} \left(E\left[\frac{g_1+d_1}{2}\right]-1\right)-g_1 < \frac{d_1+g_1}{2}-g_1 = \frac{d_1-g_1}{2} < \frac{1}{2}\frac{d_0-g_0}{2} < \frac{n}{2^2}$$

ou

$$d_2-g_2 \overset{g_2=m+1}{=} d_1-\left(E\left[\frac{g_1+d_1}{2}\right]+1\right) < d_1-\frac{g_1+d_1}{2} = \frac{d_1-g_1}{2} =< \frac{1}{2}\frac{d_0-g_0}{2} < \frac{n}{2^2}$$

- **à fortiori au rang k:**

$$d_k-g_k < \frac{d_{k-1}-g_{k-1}}{2} < \frac{d_{k-2}-g_{k-2}}{2^2} < \frac{d_0-g_0}{2^k} < \frac{n}{2^k}$$

$\Rightarrow$ on prouve la proposition demandée.

..... et la seconde par récurrence, un peu plus rapide:

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:
$d_1-g_1 \overset{g_1=m+1}{=} \ldots$
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 $\begin{cases} g_{k-1}\leq d_{k-1} \\ T[g_{k-1}]\leq e\leq T[d_{k-1}]\end{cases}$
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 $g_k\leq d_k$ puisque l'on est entré dans la boucle.

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

$$T[g_k]\leq e\leq T[d_k]$$

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

$$T[g_k]\leq e\leq T[d_k]$$

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

Exercice n°10: Fusion ordonnée de deux tableaux

Listing 12:

1.
```python
def tri_select(T):
    n = len(T)
    for k in range(n-1,0,-1):
        max = T[0]
        rg_max = 0
        for j in range(0,k):
            if T[j] > max:
```

```python
                max=T[j]
                rg_max=j
            if T[k]<max:
                T[rg_max]=T[k]
                T[k]=max
    return T
```

La fonction est très simple avec un appel à `tri_select` pour les deux listes concaténées:

Listing 13:

```python
def class(T,U):
    return tri_select(T+U)
```

Listing 14:

2.
```python
def interclassement(T,U):
    t, u, n = len(T), len(U), t+u
    R = np.array(np.ones(n))
    i, j, k = 0, 0, 0
    while (i < t) and (j < u):
        if T[i] < U[j]:
            R[k]=T[i]
            i, k = i+1,k+1
        else:
            R[k]=U[j]
            j,k=j+1,k+1
    if i == t:
        while j < u:
            if T[t]<U[j]:
                R[k]=T[t]
                j,k=j+1,k+1
            else:
                R[k]=U[j]
                j,k=j+1,k+1
    else:
        while i < t:
            if T[i]>U[u]:
                R[k]=U[u]
                i,k=i+1,k+1
            else:
                R[k]=T[i]
                i,k=i+1,k+1
    return R
```

Exercice n°11: Modèle de déplacement des dunes par automates cellulaires (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 15:

```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 piles chacune représentée par sa hauteur h. Initialement, les piles sont toutes de hauteur nulle $h = 0$:

Listing 16:

```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 17:

```python
def actualise(piles,perdus):
        N=len(piles)-1 # nombre de piles sauf la P+1 (qui restera à 0)
        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 18:

```
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 19:

```
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
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()
```