Examen Programmation ENSAE première année 2008

Examen écrit (1 heure)

Tous documents autorisés.

L'énoncé était long et prévu plutôt pour la durée d'une heure et demi. J'ai finalement attribué 2 points à l'exercice 3 lorsque la réponse était complète et 1 point à chacune des questions comptées pour 0.5 point.

0.0.1 Calcul d'un arrondi *

Enoncé

Ecrire une fonction qui arrondit un nombre réel à 0.5 près ? Ecrire une autre fonction qui arrondit à 0.125 près ? Ecrire une fonction qui arrondit à r près où r est un réel ? Répondre uniquement à cette dernière question suffit pour répondre aux trois questions. (1 point)

Correction

Pour arrondi à 1 près, il suffit de prendre la partie entière de $x+0.5$, arrondir x à 0.5 près revient à arrondi $2x$ à 1 près.

```
def arrondi_05 (x) :
    return float (int (x * 2 + 0.5)) / 2

def arrondi_0125 (x) :
    return float (int (x * 8 + 0.5)) / 8

def arrondi (x, p) :
    return float (int (x / p + 0.5)) * p
```

fin exo 0.0.1 □

0.0.2 Même programme avec 1,2,3 boucles **

Enoncé

Le programme suivant affiche toutes les listes de trois entiers, chaque entier étant compris entre 0 et 9.

```
for a in range (0, 10) :
    for b in range (0, 10) :
        for c in range (0, 10) :
             print [a,b,c]
```

1) Proposez une solution avec une boucle `while` et deux tests `if`. (1 point)

2) Proposez une solution avec deux boucles `while`. (1 point)

Correction

1) Une seule boucle contrôle les indices a, b, c. Quand un indice atteint sa limite, on incrémente le suivant et on remet l'indice à 0.

```
a,b,c = 0,0,0
while c < 10 :
    print [a,b,c]
    a += 1
    if a == 10 :
        b += 1
        a = 0
        if b == 10 :
            c += 1
            b = 1
```

2) L'avantage de cette dernière solution est qu'elle ne dépend pas du nombre d'indices. C'est cette solution qu'il faut préconiser pour écrire une fonction dont le code est adapté quelque soit la valeur de n.

```
l = [0,0,0]
while l [-1] < 10 :
    print l
    l [0] += 1
    i = 0
    while i < len (l)-1 and l [i] == 10 :
        l [i] = 0
        l [i+1] += 1
        i += 1
```

Ce problème était mal posé : il n'est pas difficile d'introduire des tests ou des boucles redondantes ou d'enlever une des conditions d'une boucle `while` pour la remplacer un test relié à une sortie de la boucle.
fin exo 0.0.2 □

0.0.3 Suite récurrente (Fibonacci) **

Enoncé

La fonction `fibo` retourne la valeur de la suite de Fibonacci pour tout entier `n`. Quel est son coût en fonction de $O(n)$? (1 point)

```
def fibo (n) :
    if n <= 2 : return 2
    else : return fibo (n-1) + fibo (n-2)
```

Correction

Lorsqu'on cherche à calculer `fibo(n)`, on calcule `fibo(n − 1)` et `fibo(n − 2)` : le coût du calcul `fibo(n)` est égal à la somme des coûts des calculs de `fibo(n − 1)` et `fibo(n − 2)` plus une addition et un test. Le coût de la fonction `fibo(n)` est plus facile à définir par récurrence. Le coût c_n du calcul de `fibo(n)` vérifie donc :

$$c_0 = c_1 = c_2 = 1 \tag{0.1}$$
$$c_n = c_{n-1} + c_{n-2} + 2 \tag{0.2}$$

Le terme 1 dans (0.1) correspond au premier test. Le terme 2 dans (0.2) correspond au test et à l'addition. Le coût du calcul de `fibo(n)` est égal à une constante près à une suite de Fibonacci. Le code suivant permet de le vérifier en introduisant une variable globale. On modifie également la fonction `fibo` de façon à pouvoir changer u_0 et u_1.

```
nb = 0  # variable globale
def fibo (n,p) :
    global nb
    if n <= 2 :
        nb += 1
        return p # plus de récurrence
    else :
        nb += 2
        return fibo (n-1,p) + fibo (n-2,p)

for n in range (1, 20) :
    nb = 0    # remis à zéro, à chaque fois
              # nb est la mesure du coût
    print fibo(n,3)-2, nb  # nombres identiques
    # nb vérifie la récurrence de la suite c(n)
    #          c(n) = c(n-1) + c(n-2) + 2
```

suites de la forme (0.3) :

$$
\begin{aligned}
u_0 &= p \\
u_1 &= p \\
u_2 &= u_0 + u_1 + d = 2p + d \\
u_3 &= u_1 + u_2 + d = 3p + 2d \\
u_4 &= u_2 + u_3 + d = 5p + 4d \\
u_5 &= u_3 + u_4 + d = 8p + 7d \\
u_6 &= u_4 + u_5 + d = 13p + 12d \\
u_7 &= u_5 + u_6 + d = 21p + 20d \\
u_8 &= u_6 + u_7 + d = 34p + 33d \\
&\dots
\end{aligned}
$$

Dans cet exemple, on s'aperçoit que la suite est égale à son coût. Pour le démontrer d'une façon plus théorique, on s'intéresse aux suites de la forme suivante dont la récurrence est développée ci-dessus à droite :

$$
\begin{aligned}
u_0 &= u_1 = u_2 = p \\
u_n &= u_{n-1} + u_{n-2} + d
\end{aligned}
\tag{0.3}
$$

Par une astuce de calcul, on peut réduire l'écriture de la suite (u_n) à une somme linéaire de deux suites (U_n) et (V_n). La suite (U_n) est définie par $U_0 = 1$, $U_1 = 1$, $U_n = U_{n-1} + U_{n-2}$ et la suite (V_n) définie par $V_0 = 0$, $V_1 = 0$, $V_n = V_{n-1} + V_{n-2} + 1$. On en déduit que :

$$u_n = U_n p + V_n d$$

Si on arrive à montrer que $U_n = V_n + 1$, cela montrera que la fonction `fibo` citée dans la correction est bien égale à son coût. Or :

$$V_n + 1 = V_{n-1} + V_{n-2} + 1 + 1 = (V_{n-1} + 1) + (V_{n-2} + 1)$$

La suite $V_n^* = V_n + 1$ suit bien la récurrence de la suite U_n. On vérifie que les premières valeurs sont identiques et cela montre que $U_n = V_n + 1$. On en déduit que :

$$u_n = U_n p + (U_n - 1)d = U_n(p + d) - d$$

Pour conclure, une suite de Fibonacci vérifie la récurrence $u_n = u_{n-1} + u_{n-2}$. Si on pose $\lambda_1 = \frac{1+\sqrt{5}}{2}$ et $\lambda_2 = \frac{1-\sqrt{5}}{2}$. Cette suite peut s'écrire sous la forme $u_n = A\lambda_1^n + B\lambda_2^n$. Le second terme étant négligeable par rapport au premier, le coût de la fonction `fibo` est donc en $O(\lambda_1^n)$. En d'autres termes, si on calcule le coût du calcul récursif d'une suite de Fibonacci, ce dernier est équivalent à la suite elle-même.

fin exo 0.0.3 □

0.0.4 Calculer le résultat d'un programme *

Enoncé

1) Qu'affiche le code suivant : (1 point)

```
l = [0,1,2,3,4,5]
g = l
for i in range (0, len (l)-1) :
    g [i] = g [i+1]
print l
print g
```

2) Et celui-ci : (1 point)

```
l = [0,1,2,3,4,5]
g = [0,1,2,3,4,5]
for i in range (0, len (l)-1) :
    g [i] = g [i+1]
print l
print g
```

3) Et encore celui-là : (1 point)

```
l = [0,1,2,3,4,5]
g = [0,1,2,3,4,5]
for i in range (0, len (l)) :
    g [i] = g [(i+1)%len (l)]
print l
print g
```

4) A votre avis, quel est l'objectif du programme et que suggérez-vous d'écrire ? (1 point)

5) Voyez-vous un moyen d'écrire plus simplement la seconde ligne `g` $= [0, 1, 2, 3, 4, 5]$ tout en laissant inchangé le résultat ? (1 point)

Correction

1) L'instruction `g` $=$ `l` implique que ces deux variables désignent la même liste. La boucle décale les nombres vers la gauche.

```
[1, 2, 3, 4, 5, 5]
[1, 2, 3, 4, 5, 5]
```

2) L'instruction `g` $= [0, 1, 2, 3, 4, 5]$ implique que ces deux variables ne désignent plus la même liste. L'instruction `print l` affiche le contenu du début.

```
[0, 1, 2, 3, 4, 5]
[1, 2, 3, 4, 5, 5]
```

3) La boucle s'intéresse cette fois-ci au déplacement du premier élément en première position. Le programmeur a pris soin d'utiliser le modulo, $n\%n = 0$ mais le premier élément de la liste g est devenu le second. Le résultat est donc :

```
[0, 1, 2, 3, 4, 5]
[1, 2, 3, 4, 5, 1]
```

4) Le programme souhaite décaler les éléments d'une liste vers la gauche, le premier élément devenant le dernier.

```
l = [0,1,2,3,4,5]
g = [0,1,2,3,4,5]
for i in range (0, len (l)) :
    g [i] = l [(i+1)%len (l)] # ligne modifiée, g devient l
print l
print g
```

5) La seconde ligne impose de répéter le contenu de la liste. Il existe une fonction qui permet de le faire :

```
import copy
l = [0,1,2,3,4,5]
g = copy.copy (l) # on pourrait aussi écrire g = list (l)
                  # ou encore g = [ i for i in l ]
for i in range (0, len (l)) :
    g [i] = l [(i+1)%len (l)]
print l
print g
```

fin exo 0.0.4 □

0.0.5 Comprendre une erreur de logique **

Enoncé

1) Un individu a écrit la fonction suivante, il inclut un seul commentaire et il faut deviner ce qu'elle fait. Après exécution, le programme affiche 4. (1 point)

```
def mystere (l) :
    """cette fonction s'applique à des listes de nombres"""
    l.sort ()
    nb = 0
    for i in range (1,len (l)) :
        if l [i-1] != l [i] : nb += 1
    return nb+1

l = [4,3,1,2,3,4]
print mystere (l)  # affiche 4
```

2) Après avoir écrit l'instruction `print l`, on s'aperçoit que la liste `l` a été modifiée par la fonction `mystere`. Qu'affiche cette instruction ? (1 point)

3) Comment corriger le programme pour que la liste ne soit pas modifiée ? (1 point)

Correction

1) La fonction travaille sur une liste de nombres triés. La fonction parcourt cette liste et compte le nombre de fois que la liste triée contient deux éléments successifs différents. La fonction retourne donc le nombre d'éléments distincts d'une liste.

2) La fonction trie la liste, elle ressort donc triée de la fonction car le passage des listes à une fonction s'effectue par adresse. Le résultat de l'instruction `print l` est donc :

```
[1, 2, 3, 3, 4, 4]
```

3) Il suffit d'utiliser le module `copy`.

```
import copy
def mystere (l) :
    """cette fonction s'applique à des listes de nombres"""
    l = copy.copy (l) # ligne insérée
                      # on peut écrire aussi l = list (l)
    l.sort ()
    nb = 0
    for i in range (1,len (l)) :
        if l [i-1] != l [i] : nb += 1
    return nb+1
```

fin exo 0.0.5 □

0.0.6 Héritage **

Enoncé

```
class Personne :
    def __init__ (self, nom) :
        self.nom = nom
    def entete (self) :
        return ""
    def __str__ (self) :
        s = self.entete () + self.nom
        return s

class Homme (Personne) :
    def __init__ (self, nom) :
        Personne.__init__ (self, nom)
    def entete (self) :
        return "M. "

class Femme (Personne) :
    def __init__ (self, nom) :
        Personne.__init__ (self, nom)
    def entete (self) :
        return "Melle "

h = Homme ("Hector")
f = Femme ("Gertrude")
print h
print f
```

Programme A

```
class Personne :
    def __init__ (self, nom, entete) :
        self.nom = nom
        self.entete = entete
    def __str__ (self) :
        s = self.entete + self.nom
        return s

h = Personne ("Hector", "M. ")
f = Personne ("Gertrude", "Melle ")
print h
print f
```

Programme B

1) Les deux programmes précédents affichent-ils les mêmes choses ? Qu'affichent-ils ? (1 point)

2) On souhaite ajouter une personne hermaphrodite, comment modifier chacun des deux programmes pour prendre en compte ce cas ? (1 point)

3) Quel programme conseilleriez-vous à quelqu'un qui doit manipuler cent millions de personnes et qui a peur de manquer de mémoire ? Justifiez. (1 point)

Correction

1) Les programmes affichent la même chose et ils affichent :

```
M. Hector
Melle Gertrude
```

2)

```
class Personne :
    def __init__ (self, nom) :
        self.nom = nom
    def entete (self) :
        return ""
    def __str__ (self) :
        s = self.entete () + self.nom
        return s

class Homme (Personne) :
    def __init__ (self, nom) :
        Personne.__init__ (self, nom)
    def entete (self) :
        return "M. "

class Femme (Personne) :
    def __init__ (self, nom) :
        Personne.__init__ (self, nom)
    def entete (self) :
        return "Melle "

class Hermaphrodite (Personne) :
    def __init__ (self, nom) :
        Personne.__init__ (self, nom)
    def entete (self) :
        return "Melle et M. "

h = Homme ("Hector")
f = Femme ("Gertrude")
g = Hermaphrodite ("Marie-Jean")
print h
print f
print g
```

Programme A

```
class Personne :
    def __init__ (self, nom, entete) :
        self.nom = nom
        self.entete = entete
    def __str__ (self) :
        s = self.entete + self.nom
        return s

h = Personne ("Hector", "M. ")
f = Personne ("Gertrude", "Melle ")
g = Personne ("Marie-Jean", \
                    "Melle et M. ")
print h
print f
print g
```

Programme B

3) Les deux programmes définissent une personne avec un prénom et un en-tête. Dans le second programme, l'en-tête est un attribut de la classe et cette classe permet de modéliser les hommes et les femmes. Dans le premier programme, l'en-tête est défini par le type de la classe utilisée. Il y a moins d'information mémorisée, en contre partie, il n'est pas possible d'avoir d'autres en-têtes que `M.` et `Melle`. Pour manipuler 100 millions de personnes, il vaut mieux utiliser le premier programme avec deux classes, il sera moins gourmant en mémoire.

fin exo 0.0.6 □

0.0.7 Analyser un programme ***

Enoncé

On considère la fonction suivante qui prend comme entrée une liste d'entiers. Elle ne retourne pas de résultat car elle modifie la liste. Appeler cette fonction modifie la liste donnée comme paramètre. Elle est composée de quatre groupes d'instructions.

```
def tri_entiers(l):
    """cette fonction s'applique à une liste d'entiers"""

    # groupe 1
    m = l [0]
    M = l [0]
    for k in range(1,len(l)):
        if l [k] < m : m = l [k]
        if l [k] > M : M = l [k]

    # groupe 2
    p = [0 for i in range (m,M+1) ]
    for i in range (0, len (l)) :
        p [ l [i] - m ] += 1

    # groupe 3
```

```
R     = [0 for i in range (m,M+1) ]
R [0] = p [0]
for k in range (1, len (p)) :
    R [k] = R [k-1] + p [k]

# groupe 4
pos = 0
for i in range (1, len (l)) :
    while R [pos] < i : pos += 1
    l [i-1] = pos + m
l [len (l)-1] = M
```

1) Que fait le groupe 1 ? (0,5 point)

2) Que fait le groupe 2 ? (0,5 point)

3) Que fait le groupe 3 ? (0,5 point)

4) Que fait le groupe 4 ? (1 point)

5) Quel est la boucle contenant le plus grand nombre d'itérations ? (0,5 point)

Correction

1) Le groupe 1 détermine les minimum et maximum de la liste `l`.

2) Dans le groupe 2, la liste `p` compte le nombre d'occurrences d'un élément dans la liste `l`. `p[0]` correspond au nombre de fois que le minimum est présent dans la liste, `p[len(l) - 1]` correspond au nombre de fois que le maximum est présent.

3) Le groupe 3 construit la fonction de répartition de la liste `l`. Si `m` est le minimum de la liste `l`, `R[i]` est le nombre d'éléments inférieurs ou égaux à `i + m` que la liste contient.

4) Le dernier groupe trie la liste `l`. Si X est une variable aléatoire de fonction de répartition F alors $F(X)$ suit une loi uniforme sur $[0, 1]$. C'est le même principe ici. La figure 0.1 illustre la fonction de répartition. A chaque marche correspond un élément de la liste et à chaque marche correspond une case du tableau `R`. En parcourant les marches du tableau `R`, on retrouve les éléments de la liste `l` triés. La marche plus haute traite le cas de plusieurs éléments égaux.

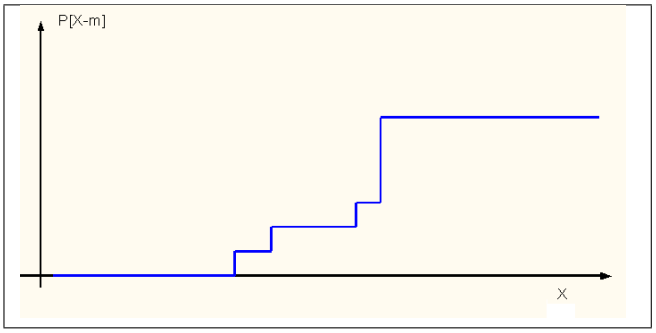

Figure 0.1 : *Construction de la fonction de répartition.*

5) Le groupe 1 inclut une boucle de n itérations où n est le nombre d'éléments de la liste `l`. Les groupes 2 et 3 incluent une boucle de $M - n$ itérations. Le groupe 4 inclut deux boucles, la première inclut n itérations, la seconde implique une variable `pos` qui passe de m à M. Le coût de ce bloc est en $O(n + M - n)$. C'est ce dernier bloc le plus long.

fin exo 0.0.7 □

0.0.8 Comprendre une erreur de logique ***

Enoncé

Chaque année, une société reçoit des cadeaux de ses clients et son patron souhaite organiser une tombola pour attribuer ces cadeaux de sorte que les plus bas salaires aient plus de chance de recevoir un cadeau.

Il a donc choisi la méthode suivante :

- Chaque employé considère son salaire net par mois et le divise par 1000, il obtient un nombre entier n_i.
- Dans une urne, on place des boules numérotées que l'on tire avec remise.
- Le premier employé dont la boule est sortie autant de fois que n_i gagne le lot.

La société a distribué plus de 100 cadeaux en quelques années mais jamais le patron n'en a gagné un seul, son salaire n'est pourtant que cinq fois celui de sa sécrétaire. Son système est-il équitable ? Il fait donc une simulation sur 1000 cadeaux en *Python* que voici. Sa société emploie quatre personnes dont les salaires sont $[2000, 3000, 5000, 10000]$.

```
import random

def tirage (poids) :
    nb = [ 0 for p in poids ]
    while True :
        i = random.randint (0, len (poids)-1)
        nb [i] += 1
        if nb [i] == poids [i] :
            return i

salaire = [ 10000, 5000, 3000, 2000 ]
poids   = [ int (s / 1000) for s in salaire ]
nombre  = [ 0 for s in salaire ]

for n in range (0,1000) :
    p = tirage (poids)
    nombre [p] += 1

for i in range (0, len (poids)) :
    print "salaire ", salaire [i], " : nb : ", nombre [i]
```

Les résultats sont plus que probants :

```
salaire  10000  : nb :  0
salaire  5000  : nb :  49
salaire  3000  : nb :  301
salaire  2000  : nb :  650
```

Il n'a aucune chance de gagner à moins que son programme soit faux. Ne s'en sortant plus, il décide d'engager un expert car il trouve particulièrement injuste que sa secrétaire ait encore gagné cette dernière bouteille de vin.

1) Avez-vous suffisamment confiance en vos connaissances en *Python* et en probabilités pour déclarer le programme correct ou incorrect ? Deux ou trois mots d'explications seraient les bienvenus. (1 point)

2) Le patron, après une folle nuit de réflexion, décida de se débarrasser de son expert perdu dans des probabilités compliquées et modifia son programme pour obtenir le suivant :

```
import random

def tirage (poids, nb) :
    while True :
        i = random.randint (0, len (poids)-1)
        nb [i] += 1
        if nb [i] % poids [i]  == 0 :
            return i

salaire = [ 10000, 5000, 3000, 2000 ]
poids   = [ int (s / 1000) for s in salaire ]
nombre  = [ 0 for s in salaire ]
temp    = [ 0 for s in salaire ]

for n in range (0,1000) :
    p = tirage (poids, temp)
    nombre [p] += 1

for i in range (0, len (poids)) :
    print "salaire ", salaire [i], " : nb : ", nombre [i]
```

Et le résultat est beaucoup plus conforme à ses attentes :

```
salaire  10000  : nb :  90
salaire  5000  : nb :  178
salaire  3000  : nb :  303
salaire  2000  : nb :  429
```

Quelle est la modification dans les règles qu'il a apportée ? Vous paraissent-elles justes ? (1 point)

Correction

1) Le programme est correct, ses résultats aussi. Prenons le cas simple où il n'y a que le patron et sa sécrétaire. Le patron gagne 10000 euros et sa secrétaire seulement 1000 euros. On reformule le problème. On construit ensuite une suite de 10 nombres choisis uniformément au hasard dans l'ensemble $\{0, 1\}$. Le patron gagne si les 10 nombres sont les siens. Ce jeu est équivalent à celui décrit dans l'énoncé à ceci près qu'on continue le tirage même si quelqu'un gagne. Et avec cette nouvelle présentation, on peut conclure qu'il y a exactement 2^{10} tirages possibles et qu'il n'y a qu'un seul tirage gagnant pour le patron. La probabilité de gagner est seulement de 2^{-10} et non $\frac{1}{10}$ comme le patron le souhaitait.

Si on revient au jeu à quatre personnes et les salaires de l'énoncé, la probabilité de gagner du patron est cette fois de 4^{-10}, celle de sa sécrétaire est au moins supérieure à $\frac{1}{16}$ qui correspond à sa probabilité de gagner en deux tirages. Si le patron ne gagne pas, c'est que sa probabilité de gagner est beaucoup trop petite par rapport au nombre de tirages. Il en faudrait au moins 4^{-10} pour avoir une probabilité non négligeable que le patron gagne au moins une fois.

2) $90 * 5 = 450$ qui n'est pas très loin de 429 ou encore $178/2 * 5 = 445$. La secrétaire paraît avoir 5 fois plus de chances de gagner que son patron et 5/2 fois plus que la personne payée 5000 euros.

La fonction `tirage` reçoit un nouvel élément `nb` qui est un compteur. Le programme ne remet plus à zéro ce compteur entre deux tirages. Cela signifie que si la boule du patron est sortie trois fois alors que la secrétaire a gagné. Lors du prochain tirage, le compteur du patron partira de 3 et celui de sa secrétaire de 0.

Si on ne remet jamais les compteurs à zéro, au bout de 1000 tirages (et non 1000 cadeaux), la boule du patron est sortie environ autant de fois que celle de la secrétaire, soit 250 fois puisqu'il y a quatre employés. La secrétaire aura gagné $\frac{250}{2} = 125$ fois, le patron aura gagné $\frac{250}{10} = 25$ soit 5 fois moins [1].

fin exo 0.0.8 □

1. Je cite ici la plus belle des réponses retournée par un élève : *Les patrons qui organisent des tombolas sont toujours mariés à leur secrétaire donc la question ne se pose pas.*

Index

Table des matières